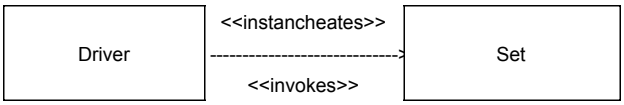
Thomas Belloli CS101
Design for Project 5



+	Driver
+	main(args:String [])

+	Set
-	int[]: set
+	Set()
+	makeEmpty()
+	isEmpty(): boolean
+	add(value: int)
+	remove(value: int)
+	elementOf(value: int): boolean
+	size(): int
+	union(setObject: set): Set
+	intersection(setObject: set): Set
+	setDifference(setObject: set): Set
+	toString(): String

Thomas Belloli CS101
Design for Project 5

Data Table for Set()

Variable or Constant	Type	Purpose

Data Table for makeEmpty()

Variable or Constant	Type	Purpose
set	int[]	to make the instance variable set, empty

Data Table for isEmpty()

Variable or Constant	Type	Purpose

Data Table for add(value: int)

Variable or Constant	Type	Purpose
tempSet	int[]	to hold value of original array
index	int	to count loops
value	int	the value to be added

Data Table for remove(value: int)

Variable or Constant	Type	Purpose
tempSet	Set	empty set for getting rid of original array value
index	int	to count loops
value	int	the value to be removed

Data Table for elementOf()

Variable or Constant	Type	Purpose
value	int	the value to be tested for

Data Table for makeEmpty()

Variable or Constant	Type	Purpose

Data Table for union(setObject: Set): Set

Variable or Constant	Type	Purpose
unionSet	Set	holds the whole range of values that is combined from the two original sets
index	int	to count loops
setObject	Set	the argument Set

Data Table for intersection(setObject: Set): Set

Variable or Constant	Type	Purpose
intersectionSet	Set	holds the values that both orignial sets have in common
index	int	to count loops
setObject	Set	the argument Set

Data Table for setDifference(setObject: Set): Set

Variable or Constant	Type	Purpose
differenceSet	Set	holds the values that neither of the original sets have in common with each other
index	int	to count loops
setObject	Set	the argument Set

Data Table for toString(): String

Variable or Constant	Type	Purpose
string	String	holds the beginning bracket for an array
index	int	to count loops

Data Table for Driver

Variable or Constant	Type	Purpose
sets	Set[]	track the set objects

Data Table for main(String[] args)

Variable or Constant	Type	Purpose
args	String[]	peramiter, unused
inputFile	File	the files that commands are pulled in from
fileScan	Scanner	the scanner to read the in file
outFile	File	the file that output is going to
fileOutput	PrintStream	the printstream used to output to the outfile
setNumber	int	used to track set input
setNumber1	int	used to track set input for the union, differance, and intersection commands
setNumber2	int	used to track set input for the union, differance, and intersection commands
setNumber3	int	used to track set input for the union, differance, and intersection commands
newNumber	int	used to track the number of the new set that will be created
empty	boolean	used to track wether or not the set is empty for the is empty command
com	string	used to hold the command and the comments for the switch and file output respectivly

Thomas Belloli CS101
Design for Project 5

```
Algotithm for main(String[] args)throws Excption
    sets <- Set[0..99]
    File inputFile <- new File(args[0])
    Scanner fileScan <- new Scanner(inputFile)
    File outFile <- new File(args[1])
    PrintStream fileOutput <- new PrintStream(outFile)
    declare ints setNumber, setNumber1, setNumber2, setNumber3
    declare int newNumber
    while (fileScan.hasNextLine())
        switch (fileScan.next()) :
        case "C":
            setNumber <- fileScan.nextInt()
            sets[setNumber] <- new Set()
            fileOutput.println("Set " + setNumber + " has been created as a new, empty set.")
            break
        case "I":
            setNumber <- fileScan.nextInt()
            empty <- true
            if (sets[setNumber] == NIL)
                fileOutput.println("There is no set " + setNumber)
            else if (empty)
                empty <- sets[setNumber].isEmpty()
                if (empty)
                    fileOutput.println("Set " + setNumber + " is empty.")
                else
                    fileOutput.println("Set " + setNumber + " is not an empty set.")
            break
        case "S":
            setNumber <- fileScan.nextInt()
            if (sets[setNumber] == NIL)
                fileOutput.println("There is no set " + setNumber)
            else if (sets[setNumber].size() > 0)
                fileOutput.println("Set " + setNumber + " contains " + sets[setNumber].size() + " values.")
            else if (sets[setNumber].isEmpty())
                fileOutput.println("Set " + setNumber + " is empty")
            break
        case "X":
            setNumber <- fileScan.nextInt()
            if (sets[setNumber] == NIL)
                fileOutput.println("There was no set " + setNumber + " to empty.")
            else if (sets[setNumber].size() >= 0)
                sets[setNumber].makeEmpty()
                fileOutput.println("The set " + setNumber + " has been emptied.")
            break
        case "A":
            setNumber <- fileScan.nextInt()
            newNumber <- fileScan.nextInt()
            if (sets[setNumber] == NIL)
                fileOutput.println("The set " + setNumber + " does not exist.")
            else if (NOT sets[setNumber].elementOf(newNumber))
                sets[setNumber].add(newNumber)
                fileOutput.println("The value " + newNumber + " has been added to the set " + setNumber + ".")
            else if (sets[setNumber].elementOf(newNumber))
                fileOutput.println("The value " + newNumber + " already exists in the set " + setNumber + ".")
```

```
        break
    case "R":
        setNumber <- fileScan.nextInt()
        newNumber <- fileScan.nextInt()
        if (sets[setNumber] == NIL)
            fileOutput.println("There is not set " + setNumber + ", in which to find the value " + newNumber + ".")
        else
            sets[setNumber].remove(newNumber)
            fileOutput.println("The value " + newNumber + " has been removed from set " + setNumber + ".")
        break
    case "F":
        setNumber <- fileScan.nextInt()
        newNumber <- fileScan.nextInt()
        if (sets[setNumber] == NIL)
            fileOutput.println("There is not set " + setNumber + ", in which to find the value " + newNumber + ".")
        else if (sets[setNumber].elementOf(newNumber))
            fileOutput.println("Set " + setNumber + " contains the value " + newNumber + ".")
        else if (NOT sets[setNumber].elementOf(newNumber))
            fileOutput.println("Set " + setNumber + " does not contain the value " + newNumber + ".")
        break
    case "U":
        setNumber1 <- fileScan.nextInt()
        setNumber2 <- fileScan.nextInt()
        setNumber3 <- fileScan.nextInt()
        if (sets[setNumber1] == NIL OR sets[setNumber2] == NIL)
            fileOutput.println("The union could not be done as one or more sets do not exist.")
        else if (sets[setNumber1] != NIL AND sets[setNumber2] != NIL)
            sets[setNumber3] <- sets[setNumber1].union(sets[setNumber2])
            fileOutput.println("Set " + setNumber3 + " is the union set of sets " + setNumber1 + " & " + setNumber2 + ".")
        break
    case "N":
        setNumber1 <- fileScan.nextInt()
        setNumber2 <- fileScan.nextInt()
        setNumber3 <- fileScan.nextInt()
        if (sets[setNumber1] == NIL OR sets[setNumber2] == NIL)
            fileOutput.println("The intersection could not be done as one or more sets do not exist.")
        else if (sets[setNumber1] != NIL AND sets[setNumber2] != NIL)
            sets[setNumber3] <- sets[setNumber1].intersection(sets[setNumber2])
            fileOutput.println("Set " + setNumber3 + " is the intersection set of sets " + setNumber1 + " & " + setNumber2 + ".")
        break
    case "D":
        setNumber1 <- fileScan.nextInt()
        setNumber2 <- fileScan.nextInt()
        setNumber3 <- fileScan.nextInt()
        if (sets[setNumber1] == NIL OR sets[setNumber2] == NIL)
            fileOutput.println("The intersection could not be done as one or more sets do not exist.")
        else if (sets[setNumber1] != NIL AND sets[setNumber2] != NIL)
            sets[setNumber3] <- sets[setNumber1].setDifference(sets[setNumber2])
            fileOutput.println("Set " + setNumber3 + " is the difference set of sets " + setNumber1 + " & " + setNumber2 + ".")
        break
    case "P":
        setNumber <- fileScan.nextInt()
        if (sets[setNumber] == NIL)
            fileOutput.println("The set " + setNumber + " does not exist, and cannot be printed.")
        else
            fileOutput.println("Set " + setNumber + " is: " + "\n\t" + sets[setNumber].toString())
        break
```

```
        case "M":
            setNumber <- fileScan.nextInt()
            newNumber <- 0
            sets[setNumber] <- new Set()
            while (fileScan.hasNextInt())
                newNumber <- fileScan.nextInt()
                sets[setNumber].add(newNumber)
            break
        case "#":
            String com <- fileScan.nextLine()
            fileOutput.println(com)
            break


Algorithm for Set()
    call makeEmpty()


Algorithm for makeEmpty()
    set <- new int[0]


Algorithm for isEmpty()
    if (size() == 0)
        return true
    return false


Algorithm for add(value: int)
    if !elementOf(value)
        tempSet <- new int[size() +1]
        for index <-0, index is less than size(), increment index by + 1
            tempSet[index] <- set[index]
        tempSet[size()] <- value
        set <- tempSet


Algorithm for remove(value: int)
    if elementOf has value passing through it
        tempSet <- new set
        for index <- 0, index is less than size(), increment index by + 1
            if set[index] is not value
                add set[index] to tempSet
        set <- tempSet.set


Algorithm for elementOf(value: int): boolean
    for index <- 0, index is less than size(), increment index by + 1
        if set[index] is equal to value
            return true
    return false


Algorithm for size()
    return set.length
```

```
Algorithm for union(setObject: Set): Set
    unionSet <- new set
    for index <- 0, index is less than size(), increment index by + 1
        add set[index] to unionSet by calling the add method
    for index <- 0, index is less than setObject.size(), increment index by + 1
        add setObject.set[index] to unionSet by calling the add method
    return unionSet


Algorithm for intersection(setObject: Set): Set
    intersectionSet <- new set
    for index <- 0, index is less than setObject.size(), increment index by + 1
        if elementOf has setObject.size[index]
            add setObject.size[index] to intersectionSet by calling the add method
    return intersectionSet


Algorithm for setDifference(setObject: Set): Set
    differanceSet <- setObject passed through union method
    intersectionSet <- setObject passed through intersection method
    differenceSet <- new set
    for index <- 0, index is less than unionSet.size(), increment indexx by + 1
        if !intersectionSet.elementOf(unionSet.set[index])
            add unionSet.set[index] to differenceSet by calling the add method
    return differenceSet


Algorithm for toString(): String
    string <- set to starting bracket
    if size() is greater than 0
        string <- set[0]
        for index <- 1, index is less than size(), increment index by + 1
            string <- string plus comma plus set[index]
    return string plus ending bracket
```