Contents

1 Basic Test Results

```
Logins: roeia1



compiling with
    javac -cp .:/cs/course/2013/oop/lib/junit4.jar  *.java oop/ex6/filescript/*.java


tests output :
        Perfect!
```

2 README

```
roeia1

####File Description:####

README - This file.

MyFileScript.java - The main that manage the entire program.

Parsing.java - In this class I divided the command file into sections.

Section.java - In this class I implemented a section.

Type2Exception.java - An exception that when it occur, the program notify with
an error and exit.

All.java - A filter with no condition.

Between.java - A size filter that checks if its between 2 values.

Contain.java - A name filter that checks if the file name contains the given
string.

Executable.java - A permission filter.

FileName.java - A name filter that checks if the file name equals to the given
string.

Filter.java - An interface containing the common method isPass for each filter.

FilterException.java - The common filter exception.

FilterFactory.java - Creating the filter according to the command.

GreaterThan.java - A size filter that checks if the file size bigger then the
value given.

Hidden.java - A permission filter.

NameFilters.java - A common filter to all the filters that checks strings.

Negative.java - A filter that negate a certain filter.

PermissionException.java - A permission exception. not a "YES"/"NO" value.

PermissionFilters.java - A common filter to all the filters that checks
permissions.

Prefix.java - A name filter that checks if the file name begin with a given
string.

SizeException.java - A size exception that gather all the exceptions in this
subject.

SizeFilters.java - A common filter to all the filters that checks sizes.

SmallerThan.java - A size filter that checks if the file size smaller then the
value given.

Suffix.java - A name filter that checks if the file name end with a given
```

```
string.

ValueOutOfRangeException.java - A size exception that indicate the value
is out of range.

Writable.java - A permission filter.

Abs.java - An order that arrange the files from a to z.

BadOrderNameException.java - An order exception when the order name is not
valid.

Compare.java - An abstract class that all the orders inherit from and
implements order, that have the sortby method.

Order.java - An interface that has the methods sortby and compare.

OrderException.java - A common order exception.

OrderFactory.java - Creating the order according to the command.

Reverse.java - An order that reverse a certain order.

Size.java - An order that arrange the files by size.

Type.java - And order that arrange the files by type from a to z.

####Design:####

I implemented the file processing with the design from tirgul10.
Also, I did hierarchy to the exceptions, filters and orders with inheritance
so that when adding something when necessary wont be hard at all.
The main method is first of all parsing the command file, dividing it into
sections and then it print the errors for each section, making a list of the
files that pass the current filter of the section and rearrange the list with
the current order of the section.

####Implementation Issues:####

I handled the different errors with ease with inheritance.
The type 1 errors I caught in the parsing phase and then i logged the warning
line number, and the type 2 errors I caught in the main method and print error.
Of course I created each of the errors in the right package that it belongs to.
I used a linked list to sort my matched files, using a comparator.
I chose this data structure because I figured it's easiest to add a matched
file and to run on the entire list with a "for each" loop.
I also made the All filter to be singleton because every object of him will
be the same, so I limit him to be only one.

####Answers to Questions:####

Addressed all the questions in previous sections.

####Source files:####

README
MyFileScript.java
Parsing.java
Section.java
Type2Exception.java
All.java
Between.java
Contain.java
Executable.java
FileName.java
Filter.java
FilterException.java
FilterFactory.java
```

```
GreaterThan.java
Hidden.java
NameFilters.java
Negative.java
PermissionException.java
PermissionFilters.java
Prefix.java
SizeException.java
SizeFilters.java
SmallerThan.java
Suffix.java
ValueOutOfRangeException.java
Writable.java
Abs.java
BadOrderNameException.java
Compare.java
Order.java
OrderException.java
OrderFactory.java
Reverse.java
Size.java
Type.java
```

3 oop/ex6/filescript/MyFileScript.java

```java
package oop.ex6.filescript;

import java.io.File;
import java.util.LinkedList;

public class MyFileScript {

    /** The source dir location in the argument */
    private static final int SOURCE_DIR_LOCATION = 0;
    /** The command file location in the argument */
    private static final int COMMAND_FILE_LOCATION = 1;
    /** The error message for type 2 error */
    private static final String ERROR_2_MESSAGE = "ERROR";

    public static void main(String[] args) {
        try {
            // Creating the sections from the command file
            LinkedList<Section> sectionList = Parsing.parser
                    (args[COMMAND_FILE_LOCATION]);

            File file = new File(args[SOURCE_DIR_LOCATION]);
            File[] fileArray = file.listFiles();
            LinkedList<File> filteredFiles;
            /*
             *  For each section checking for warnings and print the filtered
             *  files after ordering them.
             */
            for(Section currentSection : sectionList) {
                filteredFiles = new LinkedList<>();
                if(currentSection.getFilterWarningLine() !=
                        Parsing.NO_WARNING) {
                    System.out.println("Warning in line " +
                            currentSection.getFilterWarningLine());
                }
                if(currentSection.getOrderWarningLine() !=
                        Parsing.NO_WARNING) {
                    System.out.println("Warning in line " +
                            currentSection.getOrderWarningLine());
                }
                for(int currentFile=0; currentFile < fileArray.length;
                        currentFile++) {
                    if(!fileArray[currentFile].isDirectory() &&
                            currentSection.getFilter().
                            isPass(fileArray[currentFile])) {
                        filteredFiles.add(fileArray[currentFile]);
                    }
                }
                currentSection.getOrder().sortby(filteredFiles);
                printFiles(filteredFiles);
            }
        } catch (ArrayIndexOutOfBoundsException | Type2Exception e) {
            System.err.println(ERROR_2_MESSAGE);
        }
    }
    /*
     * Printing files method, getting a linked list of files and print them.
     */
    private static void printFiles(LinkedList<File> filesToPrint) {
        for(File currentFile : filesToPrint) {
            System.out.println(currentFile.getName());
        }
    }
}
```

4 oop/ex6/filescript/Parsing.java

```java
package oop.ex6.filescript;

import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.LinkedList;
import oop.ex6.filescript.filters.*;
import oop.ex6.filescript.orders.*;

public class Parsing {

    /** The value for no warning */
    protected final static int NO_WARNING = -1;
    /** The value for valid filter */
    private final static String VALID_FILTER_STRING = "FILTER";
    /** The value for valid order */
    private final static String VALID_ORDER_STRING = "ORDER";

    /** The current filter of the section */
    private static Filter currentFilter;
    /** The current order of the section */
    private static Order currentOrder;
    /** The current section in the command file */
    private static Section currentSection;
    /** The section list */
    private static LinkedList<Section> sectionList;
    /** The location of the filter line warning */
    private static int filterWarningLine;
    /** The location of the order line warning */
    private static int orderWarningLine;
    /**
     * This method running over the command file and creating a section list.
     * @param commandFile - The command file location.
     * @return A linked list of sections.
     * @throws Type2Exception - FILTER/ORDER don't appear in each section,
     * or a problem with the command file path.
     */
    public static LinkedList<Section> parser(String commandFile)
            throws Type2Exception {
        try (LineNumberReader parserReader = new LineNumberReader
                (new FileReader(commandFile));) {
            sectionList = new LinkedList<Section>();
            String currCommand = parserReader.readLine();
            // Each iteration is a section
            while (currCommand != null) {
                // Initializing no warnings
                filterWarningLine = NO_WARNING;
                orderWarningLine = NO_WARNING;
                // Checking for the filter headline
                if (!currCommand.equals(VALID_FILTER_STRING)) {
                    throw new Type2Exception();
                }
                // Getting the filter
                currCommand = parserReader.readLine();
                // Creating the filter
                try {
                    currentFilter = FilterFactory.createFilters(currCommand);
                } catch (FilterException e) {
                    currentFilter = FilterFactory.DEFAULT_FILTER;
                    filterWarningLine = parserReader.getLineNumber();
                }
                /*
                 * Checking for the order headline, if there is no order
                 * then creating the default order.
                 */
                currCommand = parserReader.readLine();
                if (currCommand == null ||
                        !currCommand.equals(VALID_ORDER_STRING)) {
                    throw new Type2Exception();
                } else {
                    // Getting and creating the order
                    currCommand = parserReader.readLine();
                    if (currCommand != null &&
                            !currCommand.equals(VALID_FILTER_STRING)) {
                        try {
                            currentOrder = OrderFactory.createOrders
                                    (currCommand);
                        } catch (OrderException e) {
                            currentOrder = OrderFactory.DEFAULT_ORDER;
                            orderWarningLine = parserReader.getLineNumber();
                        }
                        // Continue to the next section
                        currCommand = parserReader.readLine();
                    } else {
                        // Order is blank
                        currentOrder = OrderFactory.DEFAULT_ORDER;
                    }
                }
                // Creating and adding the current section to the list
                currentSection = new Section(currentFilter, currentOrder,
                        filterWarningLine, orderWarningLine);
                sectionList.add(currentSection);
            }
        } catch (IOException e) {
            throw new Type2Exception();
        }
        return sectionList;
    }
}
```

5 oop/ex6/filescript/Section.java

```java
package oop.ex6.filescript;

import oop.ex6.filescript.filters.*;
import oop.ex6.filescript.orders.*;

public class Section {

    /** The filter of the section */
    private Filter filter;
    /** The order of the section */
    private Order order;
    /** The location of the filter line warning */
    private int filterWarningLine;
    /** The location of the order line warning */
    private int orderWarningLine;
    /**
     * A data constructor.
     * @param filter - The filter of this section.
     * @param order - The order of this section.
     * @param filterWarningLine - The location of the filter line warning.
     * @param orderWarningLine - The location of the order line warning.
     */
    public Section (Filter filter, Order order,
                    int filterWarningLine, int orderWarningLine) {
        this.filter = filter;
        this.order = order;
        this.filterWarningLine = filterWarningLine;
        this.orderWarningLine = orderWarningLine;
    }

    public Filter getFilter() {
        return filter;
    }

    public Order getOrder() {
        return order;
    }

    public int getFilterWarningLine() {
        return filterWarningLine;
    }

    public int getOrderWarningLine() {
        return orderWarningLine;
    }
}
```

6 oop/ex6/filescript/Type2Exception.java

```java
package oop.ex6.filescript;

public class Type2Exception extends Exception {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

}
```

7 oop/ex6/filescript/filters/All.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class All implements Filter {

    /** The single instance of this filter */
    private static All singleAllFilter = null;
    /*
     * A private default constructor so there will be only one instance.
     */
    private All(){};
    /**
     * This method always returns a reference to the same single static All
     * filter.
     * @return The single static All filter.
     */
    public static All instance() {
        if (singleAllFilter == null) {
            singleAllFilter = new All();
        }
        return singleAllFilter;
    }
    public boolean isPass(File file) {
        // No condition to check.
        return true;
    }
}
```

8 oop/ex6/filescript/filters/Between.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class Between extends SizeFilters {

    /** The max size value */
    private double maxSizeValue;
    /**
     * A data constructor.
     * @param minSizeValue - The minimum size value.
     * @param maxSizeValue - The maximum size value.
     * @throws ValueOutOfRangeException - If the value is not between them.
     */
    public Between(double minSizeValue, double maxSizeValue)
            throws ValueOutOfRangeException {
        super(minSizeValue);
        this.maxSizeValue = maxSizeValue;
        if (this.maxSizeValue < 0 ||
                this.maxSizeValue < this.sizeValue) {
            throw new ValueOutOfRangeException();
        }
    }

    @Override
    public boolean isPass(File file) {
        double fileSize = convertToKilobytes(file.length());
        if (fileSize <= this.maxSizeValue &&
                fileSize >= this.sizeValue) {
            return true;
        } else {
            return false;
        }
    }
}
```

9 oop/ex6/filescript/filters/Contains.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class Contains extends NameFilters{

    /**
     * A data constructor.
     * @param fileName - The string being checked if contained.
     */
    public Contains(String fileName) {
        super(fileName);
    }

    @Override
    public boolean isPass(File file) {
        return file.getName().contains(this.fileName);
    }
}
```

10 oop/ex6/filescript/filters/Executable.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class Executable extends PermissionFilters {

    /**
     * A data constructor.
     * @param permissionValue - The permission value (YES/NO).
     * @throws PermissionException - If the permission value is not valid.
     */
    public Executable(String permissionValue) throws PermissionException {
        super(permissionValue);
    }

    @Override
    public boolean isPass(File file) {
        return (file.canExecute() == this.permissionValue);
    }
}
```

11 oop/ex6/filescript/filters/FileName.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class FileName extends NameFilters {

    /**
     * A data constructor.
     * @param fileName - The name to compare to the file.
     */
    public FileName(String fileName) {
        super(fileName);
    }

    @Override
    public boolean isPass(File file) {
        return file.getName().equals(this.fileName);
    }
}
```

12 oop/ex6/filescript/filters/Filter.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public interface Filter {

    /**
     * This method checks if the file passed the filter.
     * @param file - The file being checked.
     * @return The result of the check.
     */
    public boolean isPass(File file);
}
```

13 oop/ex6/filescript/filters/FilterException.java

```java
package oop.ex6.filescript.filters;

public class FilterException extends Exception {

    /**
     * 
     */
    private static final long serialVersionUID = 1L;

}
```

14 oop/ex6/filescript/filters/FilterFactory.java

```java
package oop.ex6.filescript.filters;

public class FilterFactory {

    /** The default filter */
    public final static Filter DEFAULT_FILTER = All.instance();
    /** The regex sign */
    private final static String REGEX = "#";
    /** The not command */
    private final static String NOT_COMMAND = "NOT";
    /** The filter name location in the string */
    private final static int FILTER_NAME = 0;
    /** The filter value location in the string */
    private final static int FILTER_VALUE = 1;
    /** The filter second value location in the string */
    private final static int FILTER_SECOND_VALUE = 2;
    /**
     * This method creating the filter according to the command.
     * @param filterText - The command from the command file.
     * @return The filter that created.
     * @throws FilterException - If the filter command has an error.
     */
    public static Filter createFilters(String filterText)
            throws FilterException {
        Filter filter;
        String[] filterStringArray = filterText.split(REGEX);
        switch (filterStringArray[FILTER_NAME]) {
            case "greater_than":
                filter = new GreaterThan(
                        Double.parseDouble(filterStringArray[FILTER_VALUE]));
                break;
            case "between":
                filter = new Between(
                        Double.parseDouble(filterStringArray[FILTER_VALUE]),
                        Double.parseDouble(filterStringArray
                                [FILTER_SECOND_VALUE]));
                break;
            case "smaller_than":
                filter = new SmallerThan(
                        Double.parseDouble(filterStringArray[FILTER_VALUE]));
                break;
            case "file":
                filter = new FileName(
                        filterStringArray[FILTER_VALUE]);
                break;
            case "contains":
                filter = new Contains(
                        filterStringArray[FILTER_VALUE]);
                break;
            case "prefix":
                filter = new Prefix(
                        filterStringArray[FILTER_VALUE]);
                break;
            case "suffix":
                filter = new Suffix(
                        filterStringArray[FILTER_VALUE]);
                break;
            case "writable":
                filter = new Writable(
                        filterStringArray[FILTER_VALUE]);
                break;
            case "executable":
                filter = new Executable(
                        filterStringArray[FILTER_VALUE]);
                break;
            case "hidden":
                filter = new Hidden(
                        filterStringArray[FILTER_VALUE]);
                break;
            case "all":
                filter = All.instance();
                break;
            default:
                throw new FilterException();
        }
        if (filterText.endsWith(NOT_COMMAND)) {
            filter = new Negative(filter);
        }
        return filter;
    }
}
```

15 oop/ex6/filescript/filters/GreaterThan.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class GreaterThan extends SizeFilters {

    /**
     * A data constructor.
     * @param sizeValue - The size value that being compared.
     * @throws ValueOutOfRangeException - If the size value is not valid.
     */
    public GreaterThan(double sizeValue) throws ValueOutOfRangeException {
        super(sizeValue);
    }

    @Override
    public boolean isPass(File file) {
        if (convertToKilobytes(file.length()) > this.sizeValue) {
            return true;
        } else {
            return false;
        }
    }
}
```

16 oop/ex6/filescript/filters/Hidden.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class Hidden extends PermissionFilters{

    /**
     * A data constructor.
     * @param permissionValue - The permission value (YES/NO).
     * @throws PermissionException - If the permission value is not valid.
     */
    public Hidden(String permissionValue) throws PermissionException {
        super(permissionValue);
    }

    @Override
    public boolean isPass(File file) {
        return (file.isHidden() == this.permissionValue);
    }
}
```

17 oop/ex6/filescript/filters/NameFilters.java

```java
package oop.ex6.filescript.filters;

public abstract class NameFilters implements Filter {

    /** The name value being compared to */
    protected String fileName;
    /**
     * A data constructor.
     * @param fileName - The name to compare to.
     */
    public NameFilters(String fileName) {
        this.fileName = fileName;
    }
}
```

18 oop/ex6/filescript/filters/Negative.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class Negative implements Filter {

    /** The filter being negated */
    private Filter filter;
    /**
     * A data constructor.
     * @param filter - The filter being negated.
     */
    public Negative(Filter filter) {
        this.filter = filter;
    }

    @Override
    public boolean isPass(File file) {
        return !filter.isPass(file);
    }
}
```

19 oop/ex6/filescript/filters/PermissionException.java

```
package oop.ex6.filescript.filters;

public class PermissionException extends FilterException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

}
```

20 oop/ex6/filescript/filters/PermissionFilters.java

```java
package oop.ex6.filescript.filters;

public abstract class PermissionFilters implements Filter {

    /** The value for allowed permission */
    protected final String ALLOWED = "YES";
    /** The value for denied permission */
    protected final String DENIED = "NO";

    /** The permission value */
    protected boolean permissionValue;
    /**
     * A data constructor.
     * @param permissionValue - The permission value to check.
     * @throws PermissionException - If the permission value is not valid.
     */
    public PermissionFilters(String permissionValue)
            throws PermissionException {
        if (permissionValue.equals(ALLOWED)) {
            this.permissionValue = true;
        } else if (permissionValue.equals(DENIED)) {
            this.permissionValue = false;
        } else {
            throw new PermissionException();
        }
    }
}
```

21 oop/ex6/filescript/filters/Prefix.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class Prefix extends NameFilters {

    /**
     * A data constructor.
     * @param fileName - The string being checked if starts with.
     */
    public Prefix(String fileName) {
        super(fileName);
    }

    @Override
    public boolean isPass(File file) {
        return file.getName().startsWith(fileName);
    }

}
```

22 oop/ex6/filescript/filters/SizeException.java

```java
package oop.ex6.filescript.filters;

public class SizeException extends FilterException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

}
```

23 oop/ex6/filescript/filters/SizeFilters.java

```java
package oop.ex6.filescript.filters;

public abstract class SizeFilters implements Filter {

    /** The number of bytes in kilobyte */
    private final int BYTES_IN_KILOBYTE = 1024;

    /** The size value */
    protected double sizeValue;
    /**
     * A data constructor.
     * @param sizeValue - The size value to compare.
     * @throws ValueOutOfRangeException - If the size value is negative.
     */
    public SizeFilters(double sizeValue) throws ValueOutOfRangeException {
        this.sizeValue = sizeValue;

        if (this.sizeValue < 0) {
            throw new ValueOutOfRangeException();
        }
    }
    /**
     * Converting bytes size to kilobytes size.
     * @param bytesSize - The bytes size to convert.
     * @return The kilobytes size.
     */
    protected double convertToKilobytes(double bytesSize) {
        // Return file size in bytes
        return bytesSize/BYTES_IN_KILOBYTE;
    }
}
```

24 oop/ex6/filescript/filters/SmallerThan.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class SmallerThan extends SizeFilters {

    /**
     * A data constructor.
     * @param sizeValue - The size value being compared.
     * @throws ValueOutOfRangeException - If the size value is not valid.
     */
    public SmallerThan(double sizeValue) throws ValueOutOfRangeException {
        super(sizeValue);
    }

    @Override
    public boolean isPass(File file) {
        if (convertToKilobytes(file.length()) < this.sizeValue) {
            return true;
        } else {
            return false;
        }
    }

}
```

25 oop/ex6/filescript/filters/Suffix.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class Suffix extends NameFilters {

    /**
     * A data constructor.
     * @param fileName - The string being checked if ends with.
     */
    public Suffix(String fileName) {
        super(fileName);
    }

    @Override
    public boolean isPass(File file) {
        return file.getName().endsWith(fileName);
    }

}
```

26 oop/ex6/filescript/filters/ValueOutOfRangeExceptio

```java
package oop.ex6.filescript.filters;

public class ValueOutOfRangeException extends SizeException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

}
```

27 oop/ex6/filescript/filters/Writable.java

```java
package oop.ex6.filescript.filters;

import java.io.File;

public class Writable extends PermissionFilters{

    /**
     * A data constructor.
     * @param permissionValue - The permission value (YES/NO).
     * @throws PermissionException - If the permission value is not valid.
     */
    public Writable(String permissionValue) throws PermissionException {
        super(permissionValue);
    }

    @Override
    public boolean isPass(File file) {
        return (file.canWrite() == this.permissionValue);
    }
}
```

28 oop/ex6/filescript/orders/Abs.java

```java
package oop.ex6.filescript.orders;

import java.io.File;

public class Abs extends Compare {

    /** The value for equals */
    private final int EQUALS = 0;

    @Override
    public int compare(File file1, File file2) {
        int compareValue = file1.getAbsolutePath().compareTo
                           (file2.getAbsolutePath());
        if (compareValue < 0) {
            return FIRST_FILE;
        } else if (compareValue > 0) {
            return SECOND_FILE;
        } else {
            return EQUALS;
        }
    }
}
```

29 oop/ex6/filescript/orders/BadOrderNameException

```java
package oop.ex6.filescript.orders;

public class BadOrderNameException extends OrderException {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

}
```

30 oop/ex6/filescript/orders/Compare.java

```java
package oop.ex6.filescript.orders;

import java.io.File;
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;

public abstract class Compare implements Order, Comparator<File>{
    /** The value for the first file in comparison */
    protected final int FIRST_FILE = -1;
    /** The value for the second file in comparison */
    protected final int SECOND_FILE = 1;
    /** The value to multiply to reverse the order */
    protected final int REVERSE = -1;

    public void sortby(LinkedList<File> toSort) {
        Collections.sort(toSort, this);
    }
}
```

31 oop/ex6/filescript/orders/Order.java

```java
package oop.ex6.filescript.orders;

import java.io.File;
import java.util.LinkedList;

public interface Order {

    /**
     * This method sorting the list according to the order.
     */
    public void sortby(LinkedList<File> toSort);
    /**
     * This method implements the operator compare.
     * @param file1 - The first file being compared.
     * @param file2 - The second file being compared.
     * @return The result of the comparison.
     */
    public int compare(File file1, File file2);
}
```

32 oop/ex6/filescript/orders/OrderException.java

```java
package oop.ex6.filescript.orders;

public class OrderException extends Exception {

    /**
     * 
     */
    private static final long serialVersionUID = 1L;

}
```

33 oop/ex6/filescript/orders/OrderFactory.java

```java
package oop.ex6.filescript.orders;

public class OrderFactory {

    /** The regex sign */
    private final static String REGEX = "#";
    /** The reverse command */
    private final static String REVERSE_COMMAND = "REVERSE";
    /** The order name location */
    private final static int ORDER_NAME = 0;
    /** The default order */
    public final static Order DEFAULT_ORDER = new Abs();
    /**
     * This method creating the order according to the command.
     * @param orderText - The command from the command file.
     * @return The order that created.
     * @throws OrderException - If the order command has an error.
     */
    public static Order createOrders(String orderText) throws OrderException {
        Order order;
        if (orderText == null) {
            return DEFAULT_ORDER;
        } else {
            String[] orderStringArray = orderText.split(REGEX);
            switch (orderStringArray[ORDER_NAME]) {
                case "abs":
                    order = new Abs();
                    break;
                case "type":
                    order = new Type();
                    break;
                case "size":
                    order = new Size();
                    break;
                default :
                    throw new BadOrderNameException();
            }
            if(orderText.endsWith(REVERSE_COMMAND)){
                order = new Reverse(order);
            }
            return order;
        }
    }
}
```

34 oop/ex6/filescript/orders/Reverse.java

```java
package oop.ex6.filescript.orders;

import java.io.File;

public class Reverse extends Compare {

    /** The order being reversed */
    private Order order;
    /**
     * A data constructor.
     * @param order - The order being reversed.
     */
    public Reverse(Order order) {
        this.order = order;
    }

    @Override
    public int compare(File file1, File file2) {
        return order.compare(file1, file2)*REVERSE;
    }
}
```

35 oop/ex6/filescript/orders/Size.java

```java
package oop.ex6.filescript.orders;

import java.io.File;

public class Size extends Compare {

    @Override
    public int compare(File file1, File file2) {
        if (file1.length() < file2.length()) {
            return FIRST_FILE;
        } else if (file1.length() > file2.length()) {
            return SECOND_FILE;
        } else {
            return OrderFactory.DEFAULT_ORDER.compare(file1, file2);
        }
    }
}
```

36 oop/ex6/filescript/orders/Type.java

```java
package oop.ex6.filescript.orders;

import java.io.File;

public class Type extends Compare {

    @Override
    public int compare(File file1, File file2) {
        String file1Type = file1.getName().substring
                (file1.getName().lastIndexOf(".") + 1);
        String file2Type = file2.getName().substring
                (file2.getName().lastIndexOf(".") + 1);
        if (file1Type.compareTo(file2Type) < 0) {
            return FIRST_FILE;
        } else if (file1Type.compareTo(file2Type) > 0) {
            return SECOND_FILE;
        } else {
            return OrderFactory.DEFAULT_ORDER.compare(file1, file2);
        }
    }
}
```