TIJ : Strings

1) Immutable Strings

➔ Strings are immutable. In class String every method which modifies the string actually creates and returns a brand new modified string. The original string will be left untouched.

2) Overloading ' + ' and StringBuilder

➔ + and += are the only operators in java which are overloaded for String class.

```
package chTijStrings;

public class PlusOverloading4String
{
	public static void main(String[] args)
	{
		String s = "mango";
		String str = s + "Java" + "TIJ";
		System.out.println(str);
	}
}
Output:
mangoJavaTIJ
```

➔ How above works is compiler will call StringBuilder class and uses method append( ) 3 times and finally it calls toString() to produce the results.

➔ StringBuilder class was introduced in Java 5, before it StringBuffer was used which ensured thread safety(all method were synchronized).

3. Unintended Recursion

➔ Every class in Java inherits Object class which have toString() method. With class String() its overridden so that they can produce a string representation of themselves. Even its works for collection like ArrayList as shown below.

```
package chTijStrings;

import java.util.*;

public class StringRecursion
{
	public static void main(String[] args)
	{
		ArrayList<String> al  = new ArrayList<>();

		al.add("Java");
		al.add("Strings");
		al.add("TIJ");

		System.out.println(al);
	}
}
Output:
[Java, Bruce, TIJ]
```

4. Formatting Outputs

➔ Since Java 5 two methods

I. printf()

ii. format()

have been added to both PrintStream and PrintWriter classes.

➔ Both methods are equivalent and both take single format string followed by one argument for each format specifier.

➔ When two methods are equivalent then why to create two methods? As mentioned in java docs method printf() is a convenience method especially for C users. Method printf() internally calls format() method only.

```
public PrintStream printf(String format, Object ... args)
    {
        return format(format, args);
    }

public PrintStream printf(Locale l, String format, Object ... args)
    {
        return format(l, format, args);
    }
```

➔ Second method above format string in locale style.

➔ Example

```
package chTijStrings;
import java.util.Calendar;

public class SimpleFormat
{
	public static void main(String[] args)
	{
		int x = 5;
		double y = 5.3;

		System.out.println("Row 1: ["+ x +" "+ y +"]");
		System.out.format("Row 1: [%d %f]\n", x, y);
		System.out.printf("Row 1: [%d %f]\n", x, y);
		System.out.printf("%s is best oops language%n", "Java");//%n for newline

		Calendar cal = Calendar.getInstance();
		System.out.printf("Date : %tc %n", cal);
		System.out.printf("Day : %td %n", cal);
		System.out.printf("Month : %tm%n", cal);
		System.out.printf("Year : %ty%n", cal);
	}
}
Output:
Row 1: [5 5.3]
Row 1: [5 5.300000]
Row 1: [5 5.300000]
Java is best oops language
Date : Wed Oct 31 12:48:09 IST 2018
Day : 31
Month : 10
Year : 18
```

➔ Above we can see how formatting takes place. %d is for Integer and %f for float(or double).

➔ Also one should keep in mind that format specifier and corresponding arguments should be in same order else there will be `IllegalFormatConversionException` at runtime as shown below.

```
package chTijStrings;

public class SimpleFormat
{
	public static void main(String[] args)
	{
		int x = 5;
		double y = 5.3;

		System.out.println("Row 1: ["+ x +" "+ y +"]");
		System.out.format("Row 1: [%f %d]\n", x, y);
		System.out.printf("Row 1: [%d %f]\n", x, y);
	}
}
Output:
Row 1: [5 5.3]
Row 1: [Exception in thread "main" java.util.IllegalFormatConversionException: f != java.lang.Integer
	at java.util.Formatter$FormatSpecifier.failConversion(Formatter.java:4302)
	at java.util.Formatter$FormatSpecifier.printFloat(Formatter.java:2806)
	at java.util.Formatter$FormatSpecifier.print(Formatter.java:2753)
	at java.util.Formatter.format(Formatter.java:2520)
	at java.io.PrintStream.format(PrintStream.java:970)
	at chTijStrings.SimpleFormat.main(SimpleFormat.java:11)
```

5. The Formatter Class

➔ Java's new formatting functionality is handled by Formatter class present in java.util package.

➔ The formatter class have much more to provide than printf(), because with printf() one may write to PrintStream but with Formatter one may write format string to many different places.

Constructor Summary

Constructors

Constructor and Description
Formatter() Constructs a new formatter.
Formatter(**Appendable** a) Constructs a new formatter with the specified destination.
Formatter(**Appendable** a, **Locale** l) Constructs a new formatter with the specified destination and locale.
Formatter(**File** file) Constructs a new formatter with the specified file.
Formatter(**File** file, **String** csn) Constructs a new formatter with the specified file and charset.
Formatter(**File** file, **String** csn, **Locale** l) Constructs a new formatter with the specified file, charset, and locale.

Formatter(Locale l)
Constructs a new formatter with the specified locale.

Formatter(OutputStream os)
Constructs a new formatter with the specified output stream.

Formatter(OutputStream os, String csn)
Constructs a new formatter with the specified output stream and charset.

Formatter(OutputStream os, String csn, Locale l)
Constructs a new formatter with the specified output stream, charset, and locale.

Formatter(PrintStream ps)
Constructs a new formatter with the specified print stream.

Formatter(String fileName)
Constructs a new formatter with the specified file name.

Formatter(String fileName, String csn)
Constructs a new formatter with the specified file name and charset.

Formatter(String fileName, String csn, Locale l)
Constructs a new formatter with the specified file name, charset, and locale.

➔ Example 1

```
package chTijStrings;

import java.util.Formatter;

public class FormatterClass1
{
	public static void main(String[] args)
	{

		Formatter f = new Formatter(System.out);
		f.format("%s is %d years old%n", "dog", 6);
		f.format("%s's total wait is %f%n ", "Elephant", 1595.56);
    }
}
Output:
dog is 6 years old
Elephant's total wait is 1595.560000
```

➔ Example 2

```
package chTijStrings;

import java.io.PrintStream;
import java.util.Formatter;

public class FormatterClass2
{
	private String name;
	private Formatter f;

	public FormatterClass2(String name, Formatter f)
	{
		this.name = name;
		this.f = f;
	}

	public void move(int x, int y)
```

```
	{
		f.format("%s turtle is at (%d, %d)\n", name, x,  y);
	}

	public static void main(String[] args)
	{
		PrintStream outAlias = System.out;
		FormatterClass2 tommy = new FormatterClass2("tommy", new Formatter(System.out));
		FormatterClass2 terry = new FormatterClass2("terry", new Formatter(outAlias));

		tommy.move(0,0);
		terry.move(4, 8);
		tommy.move(3, 4);
		terry.move(2, 5);
	}
}
Output:
tommy turtle is at (0, 0)
terry turtle is at (4, 8)
tommy turtle is at (3, 4)
terry turtle is at (2, 5)
```

6. Format Specifier

➔ Format specifier format is

```
%[argument_index$][flags][width][.precision]Conversion
```

a. Argument Index

➔ Argument index lets one explicitly match the given arguments with format specifiers.

➔ Example

```
package chTijStrings;

import java.util.Calendar;
import java.util.Formatter;

public class FormatSpecArgumentIndex
{
	public static void main(String[] args)
	{
		Formatter f = new Formatter(System.out);

		f.format("%d, %d, %d %n", 1,2,3);
		f.format("%3$d, %2$d, %1$d %n", 1,2,3);

		//< is Argument index for previous argument
		f.format("%d, %<d, %d %n", 1,2);
		f.format("Hex value for %d is %<x %n", 16);

		Calendar cal = Calendar.getInstance();
		f.format("Day is %td, month %<tm and year is %<ty%n", cal);
	}
}
Output:
1, 2, 3
3, 2, 1
1, 1, 2
Hex value for 16 is 10
Day is 31, month 10 and year is 18
```

b. width

➔ It is used to control the minimum size of the field.

➔ Formatter class guarantees that field will be of size defined by width by padding it with spaces if necessary.

```
package chTijStrings;

import java.util.Formatter;

public class FormatSpecifierWidth
{

	public static void main(String[] args)
	{
		String str = "Java Formatter work wonders";
		Formatter f = new Formatter(System.out);
		f.format("35 with no width :%d %n", 35);
		f.format("35 with 10 width :%10d %n", 35);
		f.format("3.4 with 10 width :%10f %n", 3.4f);

		f.format("str with no width :%s \n", str);
		f.format("str with 10 width :%10s", str);//no change in o/p
	}
}
Output:
35 with no width :35
35 with 10 width :        35
3.4 with 10 width :  3.400000
str with no width :Java Formatter work wonders
str with 10 width :Java Formatter work wonders
```

➔ Above %10d says that field should be of size 10. As in 35 there are two digits hence it is padded with 8 spaces.

➔ Also for float by default it prints 6 decimal places and 3 and (.) will take another two hence padding will be done only with 2 spaces.

c. Flags

flags	
-	left justification
#	alternate conversion format
0	pad with zeros instead of spaces
space	positive numbers are preceded by a space
+	positive numbers are preceded by a plus sign
,	numbers include grouping separators
(	negative numbers are enclosed in parentheses

➔ By default data is right justified. To make it left justified use (-) as flag.

➔ By default padding is done with spaces as seen in previous example. Spaces can be replaced with zeros using 0 for flag.

➔ 'Space' flag place one space before a positive number. Negative numbers don't have any effect of this flag.

➔ '+' flag will place + sign before all positive numbers. It won't affect negative numbers.

➔ ',' flag will be used for numbers as grouping separator.

➔ '(' flag will put negative number inside bracket. Positive number will have no effect of it.

➔ Example

```
package chTijStrings;

import java.util.Formatter;

public class FormatSpecifierFlag
{
	public static void main(String[] args)
	{
		Formatter f = new Formatter(System.out);
		f.format("Width length 10 :%10d %n", 35);
		f.format("Width length 10 with negative :%-10d %n", 35);
		f.format("Zero Padding :%010d %n", 35);
		f.format("Flag as space:% d %n", 35);//positive number are preceded by space
		//f.format("%    d %n", 35);//DuplicateFormatFlagException
		f.format("Flag as space for -ve no :% d %n", -35);//space dont affect negative nos.
		//f.format("%    d %n", -35);//DuplicateFormatFlagException
		f.format("+ flag :%+d %n", 35);
		f.format("+ flag for -ve no :%+d %n", -35);//+ flag dont affect -ve nos.
		f.format("commma flag :%,d %n", 35);
		f.format("commma flag :%,d %n", 35000);
		f.format("bracket flag :%(d %n", -35);
		f.format("bracket flag for +ve no :%(d %n", 35);
	}
}
Output:
Width length 10 :        35
Width length 10 with negative :35
Zero Padding :0000000035
Flag as space: 35
Flag as space for -ve no :-35
+ flag :+35
+ flag for -ve no :-35
commma flag :35
commma flag :35,000
bracket flag :(35)
bracket flag for +ve no :35
```

d. Precision

➔ Precision is opposite of width which is used to specify the maximum.

➔ Precision always comes after decimal point.

➔ Unlike width which is applicable for all data types and behaves the same with each, precision works differently for different data type. Check the example below.

```
package chTijStrings;

import java.util.Formatter;

public class FormatSpecifierPrecision
{
	public static void main(String[] args)
	{
```

```
            String str = "Java has rich set of libraries";
            float fno1 = 2.343f;
            float fno2 = 2.3454f;
            int i = 15;

            Formatter f = new Formatter(System.out);
            f.format("String str :%s%n", str);
            f.format("str with 10 width :%10s%n", str);
            f.format("str with 10 precision :%.10s%n", str);

            System.out.println();
            f.format("Floating pt fno1 :%f%n", fno1);
            f.format("fno1 with 2 precision :%.2f%n", fno1);
            f.format("fno1 with 4 precision :%.4f%n", fno1);

            System.out.println();
            f.format("Floating pt fno2 :%f%n", fno2);
            f.format("fno2 with 2 precision :%.2f%n", fno2);

            System.out.println();
            f.format("Integer i :%d%n", i);
            //f.format("Integer i with 2 precision: %.2d%n", args);//Err:
                                                IllegalFormatPrecisionException
      }
}
Output:
String str :Java has rich set of libraries
str with 10 width :Java has rich set of libraries
str with 10 precision :Java has r

Floating pt fno1 :2.343000
fno1 with 2 precision :2.34
fno1 with 4 precision :2.3430

Floating pt fno2 :2.345400
fno2 with 2 precision :2.35

Integer i :15
```

- ➔ sdf
- ➔ sdf
- ➔ sdf
- ➔ sdf
- ➔ sdf