Java Basics :

**wrapper classes**.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.

Creating Format Strings

You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of −

Example

System.out.printf("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

You can write −

String fs;

fs = String.format("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

System.out.println(fs);

## The void Keyword

The void keyword allows us to create methods which do not return a value.

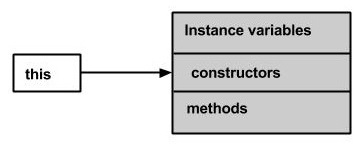
## Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method **overloading**. It is different from overriding. In **overriding**, a method has the same method name, type, number of parameters, etc.

The this keyword

**this** is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

**Note** − The keyword *this* is used only within instance methods or constructors



In general, the keyword *this* is used to −

* Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

class Student {

int age;

Student(int age) {

this.age = age;

}

}

## The finalize( ) Method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.

The finalize( ) method has this general form −

protected void finalize( ) {

// finalization code here

}

# Java - Files and I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java.

## Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

* **InPutStream** − The InputStream is used to read data from a source.
* **OutPutStream** − The OutputStream is used for writing data to a destination.

import java.io.\*;

public class CopyFile {

public static void main(String args[]) throws IOException {

FileInputStream in = null;

FileOutputStream out = null;

try {

in = new FileInputStream("input.txt");

out = new FileOutputStream("output.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

}finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

}

}

}

}

Now let's have a file **input.txt** with the following content −

This is test for copy file.

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

* **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
* **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
* **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

**Example**

[Live Demo](http://tpcg.io/lVH2u1)

import java.io.\*;

public class ReadConsole {

public static void main(String args[]) throws IOException {

InputStreamReader cin = null;

try {

cin = new InputStreamReader(System.in);

System.out.println("Enter characters, 'q' to quit.");

char c;

do {

c = (char) cin.read();

System.out.print(c);

} while(c != 'q');

}finally {

if (cin != null) {

cin.close();

}

}

}

}

### FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

File f = new File("C:/java/hello");

InputStream f = new FileInputStream(f);

## FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

OutputStream f = new FileOutputStream("C:/java/hello")

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

File f = new File("C:/java/hello");

OutputStream f = new FileOutputStream(f);

### Creating Directories

There are two useful **File** utility methods, which can be used to create directories −

* The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
* The **mkdirs()** method creates both a directory and all the parents of the directory.

## The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax −

### Syntax

try {

// Protected code

} catch (ExceptionType1 e1) {

// Catch block

} catch (ExceptionType2 e2) {

// Catch block

} catch (ExceptionType3 e3) {

// Catch block

}finally {

// The finally block always executes.

}

Listing Directories

You can use **list( )** method provided by **File** object to list down all the files and directories available in a directory as follows −

**Example**

import java.io.File;

public class ReadDir {

public static void main(String[] args) {

File file = null;

String[] paths;

try {

// create new file object

file = new File("/tmp");

// array of files and directory

paths = file.list();

// for each name in the path array

for(String path:paths) {

// prints filename and directory name

System.out.println(path);

}

} catch (Exception e) {

// if any error occurs

e.printStackTrace();

}

}

}

This will produce the following result based on the directories and files available in your **/tmp** directory −

**Output**

test1.txt

test2.txt

ReadDir.java

ReadDir.class

# Java - Regular Expressions

Java provides the java.util.regex package for pattern matching with regular expressions.

* **Pattern Class** − A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static **compile()** methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.
* **Matcher Class** − A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the **matcher()** method on a Pattern object.
* **PatternSyntaxException** − A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

public static void main( String args[] ) {

// String to be scanned to find the pattern.

String line = "This order was placed for QT3000! OK?";

String pattern = "(.\*)(\\d+)(.\*)";

// Create a Pattern object

Pattern r = Pattern.compile(pattern);

// Now create matcher object.

Matcher m = r.matcher(line);

if (m.find( )) {

System.out.println("Found value: " + m.group(0) );

System.out.println("Found value: " + m.group(1) );

System.out.println("Found value: " + m.group(2) );

}else {

System.out.println("NO MATCH");

}

}

}

This will produce the following result −

**Output**

Found value: This order was placed for QT3000! OK?

Found value: This order was placed for QT300

Found value: 0

### The start and end Methods

Following is the example that counts the number of times the word "cat" appears in the input string −

**Example**

[Live Demo](http://tpcg.io/xC7Jt9)

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

private static final String REGEX = "\\bcat\\b";

private static final String INPUT = "cat cat cat cattie cat";

public static void main( String args[] ) {

Pattern p = Pattern.compile(REGEX);

Matcher m = p.matcher(INPUT); // get a matcher object

int count = 0;

while(m.find()) {

count++;

System.out.println("Match number "+count);

System.out.println("start(): "+m.start());

System.out.println("end(): "+m.end());

}

}

}

This will produce the following result −

**Output**

Match number 1

start(): 0

end(): 3

Match number 2

start(): 4

end(): 7

Match number 3

start(): 8

end(): 11

Match number 4

start(): 19

end(): 22

### The matches and lookingAt Methods

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.

Both methods always start at the beginning of the input string. Here is the example explaining the functionality −

**Example**

[Live Demo](http://tpcg.io/pH3GXf)

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

private static final String REGEX = "foo";

private static final String INPUT = "fooooooooooooooooo";

private static Pattern pattern;

private static Matcher matcher;

public static void main( String args[] ) {

pattern = Pattern.compile(REGEX);

matcher = pattern.matcher(INPUT);

System.out.println("Current REGEX is: "+REGEX);

System.out.println("Current INPUT is: "+INPUT);

System.out.println("lookingAt(): "+matcher.lookingAt());

System.out.println("matches(): "+matcher.matches());

}

}

This will produce the following result −

**Output**

Current REGEX is: foo

Current INPUT is: fooooooooooooooooo

lookingAt(): true

matches(): false

### The replaceFirst and replaceAll Methods

The replaceFirst and replaceAll methods replace the text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurrences.

Here is the example explaining the functionality −

**Example**

[Live Demo](http://tpcg.io/hPbUKo)

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

private static String REGEX = "dog";

private static String INPUT = "The dog says meow. " + "All dogs say meow.";

private static String REPLACE = "cat";

public static void main(String[] args) {

Pattern p = Pattern.compile(REGEX);

// get a matcher object

Matcher m = p.matcher(INPUT);

INPUT = m.replaceAll(REPLACE);

System.out.println(INPUT);

}

}

This will produce the following result −

**Output**

The cat says meow. All cats say meow.

### The appendReplacement and appendTail Methods

The Matcher class also provides appendReplacement and appendTail methods for text replacement.

Here is the example explaining the functionality −

**Example**

[Live Demo](http://tpcg.io/AvfsqL)

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

private static String REGEX = "a\*b";

private static String INPUT = "aabfooaabfooabfoob";

private static String REPLACE = "-";

public static void main(String[] args) {

Pattern p = Pattern.compile(REGEX);

// get a matcher object

Matcher m = p.matcher(INPUT);

StringBuffer sb = new StringBuffer();

while(m.find()) {

m.appendReplacement(sb, REPLACE);

}

m.appendTail(sb);

System.out.println(sb.toString());

}

}

This will produce the following result −

**Output**

-foo-foo-foo-

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch**keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

### Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

[Live Demo](http://tpcg.io/gVDoOF)

// File Name : ExcepTest.java

import java.io.\*;

public class ExcepTest {

public static void main(String args[]) {

try {

int a[] = new int[2];

System.out.println("Access element three :" + a[3]);

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Exception thrown :" + e);

}

System.out.println("Out of the block");

}

}

This will produce the following result −

### Output

Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3

Out of the block