# String Processing

## Tópicos

* Búsqueda, análisis y construcción de Strings
* Búsqueda, análisis y remplazo de Strings usando expresiones regulares, patrones de expresión para coincidir limitado por punto, asterisco, más, ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ()
* Formatear Strings usando los parámetros de formato %b, %c, %d, %f y %s

## Introducción

Los objetos String son usados constantemente en aplicaciones Java. Los programas usan los String no solo en la lógica de negocio, también se utilizan para implementar funcionalidad de Logging. Por lo tanto, es muy útil entender los Strings y los conceptos relacionados.

Aparte de las operaciones comunes, buscar y analizar son dos operaciones muy útiles que son ejecutadas en los Strings. Java provee un gran soporte para lograr estas operaciones eficiente y fácilmente. Empezando por la versión 4, Java soporta expresiones regulares y ofrece dos clases útiles para explotar el poder de las expresiones regulares. Desde la versión 5, Java también soporta el método como en C printf() para formatear Strings.

El examen tiene tres temas relacionados con Strings: “Crear y manipular Strings”; “Manipular datos usando la clase StringBuilder y sus métodos”; y “Probar la igualdad entre Strings y otros objetos usando == y el método equals()”: Asumienddo que ya tienes los prerrequisitos para el examen y estás familiarizado con el uso de Strings, procederemos a los temas que cubre el OCPJP7.

Como candidato para el examen, debes de estar familiarizado con la búsqueda, análisis y construcción de Strings; expresiones regulares y su uso; y el formateo de Strings así como los especificadores de formato. En este capítulo vamos a cubrir esos temas a detalle.

## Procesando Strings

En esta sección aprenderás cómo puedes procesar caracteres almacenados en un objeto String.

### Búsqueda de String

Cuando trabajas con Strings, puedes necesitar buscar un String dado. Hay muchas versiones sobrecargadas del método indexOf() disponibles en la clase String, aparte del método lastIndexOf() para buscar un String al final, y el método regionMatches() para comparar una región de texto dentro de una String. Vamos a discutir cada uno de estos métodos uno a uno.

#### indexOf()

¿Cómo puedes buscar un carácter en particular en un String, y si está, cuál es la posición? Por ejemplo, cuál sería la posición del carácter “J” ocupado en el String “OCPJP”:

El método indexOf() busca el carácter especificado y retorna la PRIMERA ocurrencia del carácter (el índice empieza con 0 como en los arrays). Ahora vamos a buscar una sub-string a una String dada. El método indexOf() está sobre-cargado y una definición del método toma una String como argumento de búsqueda. Veamos un ejemplo:

// SearchString2.java

public class SearchString2 {

public static void main(String... args) {

String str = “I am preparing for OCPJP”;

System.out.println(“La sub-string \”for\” ocurre al índice: ” + str.indexOf(“for”));

}

}

Imprime

La sub-string “for” ocurre al índice: 15

// SearchString1.java

public class SearchString1 {

public static void main(String... args) {

String str = “OCPJP”;

System.out.println(“El carácter J ocurre en el índice: ” + str.indexOf(‘J’));

}

}

Imprime

El carácter J ocurre en el índice 3

¿Qué pasaría si la string buscada no existe en el String? Entonces retorna -1 que significa “no encontrada”.

Estos dos son problemas fáciles; vamos a intentar con uno un poco más difícil. Dada una String grande, ¿Cómo puedes encontrar cuántas veces ocurre la string “am” en esa String? Nada de qué preocuparte, en este caso hay otra versión del método indexOf(). En este método puedes especificar un índice que indica desde dónde va a comenzar la búsqueda. Vamos a ver una implementación:

// SearchString3.java

public class SearchString3 {

public static void main(String... args) {

String str = “I am a student. I am preparing for OCPJP”;

int fromIndex = 0;

while(str.indexOf(“am”, fromIndex > -1)) {

formIdex = str.indexOf(“am”, fromIndex);

System.out.println(“Substring \”am\” ocurre al índice: ” + fromIndex);

fromIndex++;

}

}

}

Imprime

Substring “am” ocurre al índice: 2

Substring “am” ocurre al índice 18

Usas un while-loop para checar cuando hay más ocurrencias de la string buscada. Esto también mantiene un índice (fromIndex) de donde iniciarás la búsqueda. Se incrementa el fromIndex después de cada ocurrencia, por lo tanto la siguiente ocurrencia puede ser encontrada.

Si quieres buscar la última ocurrencia de la string buscada, puedes usar las versiones sobre cargadas del método lastIndexOf definidas en la clase String. La clase String también tiene métodos simples para checar el principio o el final de las strings y estos métodos están en la siguiente tabla:

|  |  |
| --- | --- |
| Método | Descripción |
| boolean startsWith(String prefixString, int offset) | Empezando por el offset, checa si esta String contiene la prefixString |
| boolean startsWith(String prefixString) | Es igual que startsWith(prefixString, 0) |
| boolean endsWith(String suffixString) | Checa si esta string tiene suffixString |

regionMatches()

Considera la string “Tarzan: Hi Jane, wanna ride an elephant?\n Jane: No thanks!” ¿Cómo puedes checar si la string “Jane: No thanks!” está presente como la primera parte en la respuesta de Jane? (Nota: la respuesta de Jane es la región de texto que viene después de la nueva línea \n)

Hay muchas maneras de resolver este problema, y puedes usar el método regionMatches() aquí en función de aprender a usarlo. En la siguiente tabla se listan los dos métodos sobrecargados de regionMatches(). Para resolver este problema en la string dada puedes empezar por localizar el carácter “\n” y de esta posición, puedes buscar la string “Jane: No thanks!”, como se ve en el siguiente código muestra cómo.

|  |  |
| --- | --- |
| Método | Descripción |
| boolean regionMatches(int start, String matchingStr, int matchStartOffset, int matchLen) | Empezando por start en esta String, checa si la región del texto dato por matchingStr coincide. En matchStr, checa por el número de caracteres matchLen empezando por matchStartOffset |
| boolean regionMatches(boolean ignoreCase, int start, String matchString, int matchStartOffset, int matchLen) | Lo mismo que un método previo, pero con un argumento adicional al inicio que si es true, ignora si son altas o bajas las letras |

### Análisis de cadenas

public class MatchRegionInString {

public static void main(String[] args) {

String chat = “Tarzan: Hi Jane, wanna ride an Elephant? \n Jane: No: thanks! I’m” +

“ preparing for OCPJP now!”);

String matchString = “Jane: No thanks!”;

// primero obtenemos la posición de donde inicia la región a buscar

int startIndex = chat.indexOf(‘\n’);

System.out.println(“La respuesta de Jane empieza en el índice: ” + startIndex);

// si encuentra a \n intenta buscar la String coincidente “Jane: no thanks!”

// desde startIndex

if(startIndex > -1) {

boolean doesMatch = chat.regionMatches(startIndex + 1, matchString, 0, matchString.length());

if(doesMatch)

System.out.println(“El string coincide” + matchString);

}

}

}

Analizar cadenas es una operación útil e interesante en los Strings. (La palabra parse significa analizar para descomponer en partes constituyentes basadas en una estructura dada) Ahora te presentaremos la operación básica de parsear en esta sección; después discutiremos tópicos avanzados de parseo en este capítulo cuando revisemos expresiones regulares.

#### Conversión de cadenas

En tus programas, encontrarás constantemente la necesidad de convertir cadenas de y hacia tipos primitivos como float, int, boolean. Para convertir de un tipo primitivo a un tipo String, puedes usar el método sobrecargado valueOf() de la clase String.

Vamos a empezar con la conversión del valor entero 10 a String.

String str1 = String.valueOf(10); // modo correcto de convertir un entero a un String

Nota que la asignación directa causará un error de compilación:

String str1 = 10; // genera error de compilación

String str1 = (String) 10; // genera error de compilación

¿Qué pasa con la conversión a la inversa: si el String tiene un valor de algún tipo primitivo, ¿Cómo puedes ejecutar la conversión? Obviamente , las siguientes dos sentencias, que intentan asignar o cambiar directamente, a través de un casteo explícito, resultan en un error de compilación:

int i = “10”;

int i = (int) “10”;

Para hacer esta conversión, necesitas usar el método estático parseInt() disponible en la clase Integer como sigue:

int i = Integer.parseInt(“10”); // modo correcto de convertir un String a int

Este método está sobrecargado. Hay otra versión de este método que toma un argumento adicional: la base (o radix) del valor integral como octal o hexadecimal. Las clases envoltorio Byte, Short, Long, Float y Double tienen los métodos equivalentes de parseo para convertir strings a sus correspondientes tipos primitivos. ¿Qué pasa si tú pasas un argumento inválido a uno de estos métodos? Por ejemplo:

float f = Float.parseFloat(“no such value”); // java.lang.NumberFormatException en Runtime

#### El método split()

El siguiente código es un ejemplo de cómo dividir una sentencia e imprimir todas las palabras en el string usando el método split() de la clase String:

public class ParseString1 {

public static void main(String... args) {

String quote = “Never lend books-nobody ever returns them!”

String[] words = quote.split(“ ”); // divide la cadena basado en el delimitador “ ”

for(String word: words)

System.out.println(word);

}

}

El método split() toma como delimitador una expresión regular (se verán después). En este ejemplo tú provees un espacio en blanco como delimitador, por lo que es capaz de extraer todas las palabras en la sentencia. Nota como los caracteres “-” y “!” son parte de las palabras, ya que tú no especificaste signos de puntuación como delimitadores. Puedes revisitar este problema cuando aprendas expresiones regulares.

NOTA: el argumento del método split() es una string delimitadora, cuando es una expresión regular y pasas una sintaxis inválida obtienes una excepción PatternSyntaxException.

Ahora, asume que tienes una cadena que contiene la ruta de un directorio y quieres parsearla para imprimir los nombres de cada folder individualmente. Observa la implementación:

public class ParseString2 {

public static void main(String... args) {

String str = “c:\\work\\programs\\parser”;

String[] dirList = str.split(“\\\\”);

for(int 1 = 0; i < dirList.length; i++)

System.out.println(dirList(i));

}

}

### Expresiones regulares

Una expresión regular define un patrón de búsqueda que puede ser utilizado para ejecutar operaciones como búsqueda de cadenas y manipulación de las mismas. Una expresión regular es más que una secuencia de símbolos predefinidos especificados en una sintaxis específica que te ayuda a buscar o manipular cadenas. Una expresión regular es especificada como una cadena que se aplica a otra, de izquierda a derecha.

Puedes preguntarte para qué necesitas regex cuando puedes ejecutar una búsqueda directamente usando la función de String como viste en los temas pasados, usando indexOf(), por ejemplo. Bueno, la respuesta es muy simple. Puedes usar el método indexOf() o cualquier otro método similar cuando conoces con exactitud la cadena que vas a buscar. Sin embargo, en algunos casos donde tú solo conoces el patrón de la cadena pero no la cadena específica necesitas usar regex. Regex es una muy poderosa herramienta que simplifica la búsqueda y manipulación de Strings. Por ejemplo, digamos que necesitas buscar todas las direcciones de e-mail en una cadena. No puedes lograrlo usando el método indexOf() ya que no sabes las direcciones exactas; sin embargo, puedes usar regex para especificar un patrón que encontrará todos los e-mails en la cadena dada.

#### Entender los símbolos de regex

Ahora nos enfocaremos en entender la sintaxis y la semántica de los símbolos usados para especificar regex. En la siguiente tabla se muestran los símbolos usados comúnmente:

|  |  |
| --- | --- |
| Símbolo | Descripción |
| ^expr | Coincide la expresión al principio de la línea |
| expr$ | Coincide la expresión al final de la línea |
| . | Coincide un carácter único con excepción de nueva línea |
| [xyz] | Coincide cualquiera: x o y o z |
| [p-z] | Especifica un rango. Coincide cualquier carácter de la p a la z |
| [p-z1-9] | Coincide tanto caracteres de la p a la z como cualquier dígito del 1 al 9 (recuerda, este no quiere coincidir p1) |
| [^p-z] | ^ como primer carácter dentro de las llaves niega el patrón; coincide cualquier carácter con excepción de la p a la z |
| Xy | Coincide x seguida de y |
| x|y | Coincide x o y |

Puedes usar los símbolos dados en la tabla anterior y especificar un regex. Por ejemplo, puedes escribir “[0-9]” para coincidir todos los números (dígitos) o “[\t\r\f\n]” para coincidir todos los espacios en blanco. También puedes usar ciertos metasímbolos predefinidos para facilitar la especificación de un regex. Por ejemplo, puedes especificar “\d” en lugar de “[0-9]” para coincidir dígitos o “\s” en lugar de “[\t\r\f\n]” para coincidir con todos los espacios en blanco. La siguiente tabla muestra una lista de los metasímbolos más comunes:

|  |  |
| --- | --- |
| Símbolo | Descripción |
| \d | dígitos del 0 al 9 |
| \D | Coincide siempre y cuando no sea dígito |
| \w | Coincide caracteres en palabras |
| \W | Coincide caracteres que no sean palabras |
| \s | Espacios en blanco (equivalente a [\t\r\f\n]) |
| \S | Coincide con lo que no son espacios en blanco |
| \b | Coincide con el límite de la palabra cuando está fuera de las []\b. Coincide con el backslash cuando está dentro de [\b] |
| \B | Coincide con los no límites de las palabras |
| \A | Coincide con el principio del String |
| \Z | Coincide con el fin del String |

Ok, tan lejos de estar bien. Pero ¿Qué si quisieras especificar una regex cuando la coincidencia envuelve una ocurrencia de la cuenta de caracteres? Bueno, para estas situaciones puedes especificar símbolos de cuantificación provistos en la siguiente tabla:

|  |  |
| --- | --- |
| Símbolo | Descripción |
| expr? | Coincide con 0 o 1 ocurrencia de expr (equivalente a expr{0,1}) |
| expr\* | Coincide con 0 o más ocurrencias de expr (equivalente a expr{0,}) |
| expr+ | Coincide con 1 o más ocurrencias de expr (equivalente a expr{1,}) |
| expr{x} | Coincide con x ocurrencias de expr |
| expr{x,y} | Coincide ocurrencias entre x & y de expr |
| expr{x,} | Coincide con x o más ocurrencias de expr |

#### Expresiones regulares soportadas en Java

Java 1.4 presentó las expresiones regulares. El paquete java.util.regex soporta las expresiones regulares. Este consiste en dos clases importantes, Pattern y Matcher. Pattern representa una regex en una representación compilada, y Matcher interpreta la regex y coincide la respectiva subcadena en la cadena dada.

En este punto te puedes preguntar por qué es regex soportado por clases dedicadas como Matcher y Pattern, cuando otros métodos como split() en la clase String soportan regex. Una sola respuesta es: rendimiento. Las clases Pattern y Matcher están optimizadas mientras los métodos de la clase String no lo están.

Usa las clases Matcher y Pattern siempre que ejecutes búsqueda o reemplazo de cadenas de manera pesada: estas clases tienen métodos más eficientes que split() u otros métodos.

Ahora veremos cómo puedes usar las clases Pattern y Matcher. Primero necesitas llamar un método estático compile() en la clase Pattern para obtener una instancia de un pattern. El primer argumento de este método es una regex. Después necesitas llamar a otro método estático llamado matcher() de la clase Pattern para obtener una instancia de la clase Matcher. El método matcher() retorna un objeto Matcher. Este objeto es usado para ejecutar operaciones en la cadena de entrada.

Vamos a ver cómo usar las clases Pattern y Matcher. Asume que tienes una cadena que consiste en detalles personales (como nombre, dirección, teléfono...) de un conjunto de personas. Usarás la siguiente cadena para los ejemplos del resto de esta sección.

String str = “Danny Doo, Flat no 502, Big Apartment, Wide Road, Near Huge Milestone, Hugo-city 56010, Ph: 9876543210”, Email: danny@myworld.com. Maggy Myer, Post bag no 52, Big bank post office, Big bank city 56000, ph: 9876501234, Email: maggi07@myuniverse.com.”;

NOTA: necesitas especificar un regex usando el backslash (\); no uses el forwardslash (/) en su lugar. El compilador no te dará ningún error, sin embargo no obtendrás el comportamiento deseado.

#### Buscar y analizar con regex

Vamos a empezar con un ejemplo simple. Necesitas escribir un código que imprima todas las palabras de la string str. ¿Cómo puedes hacer esto? Bueno, ¿Recuerdas el metacaracter \w, el cual coincide con todos los símbolos que forman una palabra? Usarás “\w” junto con el cuantificador “+” para hacer “\w+”, que significa que quieres buscar todas las palabras de tamaño uno o más. Observa el siguiente código:

// Regex1.java

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Regex1 {

public static void main(String[] args) {

String str = “Danny Doo... \\ observar el str de arriba

Pattern pattern = Pattern.compile(“\\w+”);

Matcher matcher = Pattern.matcher(str);

while(matcher.find()) {

System.out.println(matcher.group());

}

}

}

Puedes ver que la regex busca todas las palabras que consisten en al menos un carácter. ¿Qué pasa cuando invocas al método compile() junto con el patrón regex de la clase Pattern. Después de esto, obtienes una instancia del Matcher llamando al método matcher de la instancia pattern. Y finalmente obtienes el resultado usando los métodos find() y group() de la clase Matcher. El método find() retorna true si existe ahí un resultado de búsqueda. El métdo group() retorna los resultados de la búsqueda, uno a uno, como una cadena.

NOTA: Usamos dos backslash en la regex (“\\w+”) porque el backslash es un carácter de escape en regex. Sin embargo, backslash también es un carácter de escape en las cadenas, lo que significa literal “\\” se interpreta como un único backslash. Esto se traduce en una salida interesante: escribimos “\\” como un único backslash en una regex, cuando será escrito como “\\\\” en un programa de Java si queremos especificar un único backslash.

En el mismo sentido, puedes buscar todos los números usando la regex “\d+”. Ahora, vamos a buscar e imprimir todos los códigos postales que aparecen en la string. Asume que los códigos siempre tienen de medida 5. Puedes lograr esto usando regex. Observa el siguiente programa:

// Regex2.java

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Regex2 {

public static void main(String[] args) {

String str ...

Pattern pattern = Pattern.compile(“\\d{5}”);

Matcher matcher = pattern.matcher(str);

while(matcher.find()) {

System.out.println(matcher.group());

}

}

}

Opsss!!! el código anterior imprime los zip pero también imprime tres números de teléfono de manera parcial, lo que era inesperado. Hmmm, para obtener solo el código zip debes especificar más propiedades para el regex. Observa el siguiente código

En el código anterior usaste el regex “\D\d{5}\D” y trabajó bien. Lo que hiciste fue especificar un no-dígito precedente seguido por un número de seis dígitos. Fácil cierto!! Bueno, aquí hay un problema en esta solución. Ek programa está imprimiendo un espacio en blanco justo antes del número de seis dígitos ya que coinciden con “\D”. Aquí te mostramos una solución elegante para este problema: puedes usar “\b” (usado para detectar los límites de las palabras). Observa si funciona el siguiente código:

// Regex4.java

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Regex4 {

public static void main(String... args) {

String str ...

Pattern pattern = Pattern.compile(“\\b\\d{5}\\b”);

Matcher matcher = pattern.matcher(str);

while(matcher.find())

System.out.println(matcher.group);

}

}

// Regex3.java

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Regex3 {

public static void main(String... args) {

String str ...

Pattern pattern = Pattern.compile(“\\D\\d{5}\\D”);

Matcher matcher = pattern.matcher(str);

while(matcher.find())

System.out.println(matcher.group);

}

}

Esto es perfecto. Similarmente, puedes buscar todos los números telefónicos, ya que la medida del número telefónico en el ejemplo es de 10.

Ahora vamos a intentar hacer algo más difícil: buscar las direcciones de e-mail. En una dirección de e-mail, la primera parte es una palabra (como la que especificas con el patrón “\w+”) seguida por un @, seguida por otra palabra y con el sufijo “.com” (por simplicidad ignoraremos otros sufijos como “.edu”). Entonces, la expresión regular que busca e-mails en el ejemplo es “\w+@\w+\.com”

Vamos a intentar con el siguiente código:

public class Regex5 {

public static void main(String[] args) {

String str ...

Pattern pattern = Pattern.compile(“\\w+@\\w+\\.com”);

Matcher matcher = pattern.matcher(str);

while(matcher.find())

System.out.println(matcher.group());

}

}

#### Reemplazar cadenas con regex

En la sección previa, intentaste buscar y analizar cadenas con regex. Puedes también manipular (modificar) cadenas con regex. Vamos a intentar reemplazar Strings ahora.

En la cadena del ejemplo, todos los números telefónicos están representados por cadenas de diez dígitos consecutivos. Ahora quieres cambiar este formato de número por XXX-XXXXXXX, en otras palabras, quieres insertar un guión medio (-) después del tercer dígito del número telefónico. Veamos el siguiente código:

import java.util.Matcher;

import java.util.Pattern;

public class Regex6 {

public static void main(String... args) {

String str ...

Pattern pattern = Pattern.compile(“(\\D)(\\d{3})(\\d{7})(\\D)”);

Matcher matcher = pattern.matcher(str);

String newStr = matcher.replaceAll(“$1$2-$3$4”);

System.out.println(newStr);

}

}

Bueno, esto trabaja como se espera. ¿Pero ahora lograrás esto? Una verdadera observación es que usaste el método replaceAll() de la clase Matcher. Sin embargo hay un concepto más importante que necesitas entender del ejemplo anterior.

Puedes formar grupos con regex. Estos grupos pueden usarse para especificar cuantificadores en el subconjunto deseado de todo el regex. Estos grupos también pueden ser usados para especificar una back-referencia. cada grupo puede referenciarse como “$n” donde “n” es un entero, entonces, para el ejemplo, el primer grupo puede referirse como “$1”, el segundo puede referenciarse como “$2” y así sucesivamente.

Aquí formas cuatro grupos, y cuando reemplazas insertas un guion medio (-) entre el segundo y el tercero. Por esto el reemplazo funciona.

Ahora, vamos a hacer algo diferente. Vamos a implementar un método para validar una dirección IP. Puedes sugerir un regex para coincidir con una ip. Digamos que dices “\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b”, donde “\b” coincide con los límites de la palabra y “\d{1,3}” es usado para especificar que esperas tres dígitos aquí. Bueno, es buen intento pero está mal. Una dirección IP válida consiste en 4 números separados por puntos, donde cada número puede ser entre 0 y 255 (ambos incluidos). Esto significa que cualquier número mayor a 255 resultaría en una IP inválida. Sin embargo, en el regex anterior escribiste cualquier número de tres dígitos (hasta más grandes de 255) y el regex coincidirá. Sin embargo está mal. El siguiente código muestra la correcta implementación de un regex para validar una IP.

La primera ip1 es una IP válida y la segunda no. El regex especificado por ti, es exitoso identificando las IPs. NO te alarmes por el largo regex; este es fácil de entender.

import java.util.regex.Pattern;

public class Regex7 {

void validateIP(String ipStr) {

String regex = “\\b((25[0-5]|2[0-4]\\d|[01]?\\d\\d?)(\\.)){3}(25[0-5]|2[0-4]\\d|[01]?\\d\\d?)\\b”;

System.out.println(ipString + “ is valid? ” + Pattern.matches(regex, ipStr));

}

public static void main(String... args) {

String ipStr1 = “255.254.188.123”;

String ipStr2 = “255.254.188.273”;

Regex7 validator = new Regex7();

validator.validateIP(ipStr1);

validator.validateIP(ipStr2);

}

}

Vamos a empezar con los símbolos de inicio y final “\b” es la marca límite de la palabra como viste antes. Ahora, vamos a ver el primer grupo “((25[0-5]|2[0-4]\d|[01]?\d\d?)(\.))” que tiene dos partes: la primera especifica el regex para un número menor a 256 y la segunda especifica el punto (.). La primer parte del primer grupo tiene 3 sub-expresiones. La primer sub-expresión “25[0-5]” especifica que el número puede ser en el rango de 250 a 255. La segunda sub-expresión “2[0-4]\d” especifica que el número puede ser del 200 al 249, y la tercera expresión “[01]?\d\d?” especifica el que el número puede estar ser desde 0 a 199. Con la expresión “{3}” repites este primer grupo tres veces. El segundo grupo es lo mismo que el primer grupo sin el punto y sin repetir.

### Formato de cadenas

Por mucho, hemos discutido cómo buscar o analizar cadenas. ¿Qué si quisieras formatear una cadena en una plantilla predefinida? Por ejemplo, asume que estás calculando el área de un círculo y quieres imprimir el área calculada con solo dos dígitos fraccionarios. Si intentas usar el método tradicional println(), imprimirá un número float enorme. Como otro ejemplo, digamos que quieres separar los dígitos de un número grande son un separador como la coma “,” para imprimirlo. En estos casos puedes usar el método estilo C printf() (print formatted) que fue incluido a partir de la versión 5 de Java.

El método printf() usa banderas para formatear cadenas. Es muy similar a la función printf() de C. Este método está provisto por la clase PrintStream. Aquí está la signatura:

PrintStream printf(String format, Object... args)

El primer parámetro del método es el formato de la cadena. Un formato de cadena puede contener literales String y especificadores de formato. Los argumentos son pasados como el segundo argumento (args). Este método puede lanzar una excepción IllegalFormatException si el formato pasado no es correcto.

Los especificadores de formato son la neta de los conceptos de formatear cadenas. Estos definen un marcador de posición para un tipo de dato específico y su formato (como alineación y ancho). Los parámetros restantes del método son las variables o literales que proveen los datos que van a ser formateados y los cuales ocuparán los marcadores de posición definidos en el primer parámetro.

#### Especificadores de formato

Vamos a revisar la plantilla de los especificadores de formato en el método printf():

%[flags][width][.precision]datatype\_specifier

Como puedes ver, cada especificador de formato empieza con el signo de “%” seguido de las flags, width y la información de precisión. En esta cadena, las flags, width y la información de precisión son opcionales, mientras que el “%” y el tipo de dato son obligatorios.

Las flags son símbolos de un solo carácter que especifican características como la alineación y el rellenado de los caracteres. Por ejemplo, la flag “-” especifica alineación a la izquierda, “^” especifica alineación al centro y “0” rellena el número con ceros a la izquierda.

El especificador width indica el mínimo número de caracteres que abarcarán en la cadena final formateada. Si los datos de entrada son más pequeños que width, entonces esto es rellenado con espacios por default. En caso de que los datos de entrada sean más grandes que width, los datos completos serán impresos sin que se rebanen.

El campo de precisión especifica el número de dígitos de precisión en la cadena de salida. Este campo opcional es muy útil con números de punto flotante.

Finalmente, el especificador de tipo de dato indica el tipo de dato esperado. El campo es un marcador de posición para el tipo de dato especificado en los datos de entrada. La siguiente tabla muestra los especificadores de tipo de dato comúnmente usados:

|  |  |
| --- | --- |
| Símbolo | Descripción |
| %b | Boolean |
| %c | Character |
| %d | Decimal integer (con signo) |
| %e | Número Floating point en formato científico |
| %f | Número Floating point en formato decimal |
| %g | Número Floating point en decimal o científico (depende del valor pasado) |
| %h | Código hash del argumento pasado |
| %n | Separador de línea (carácter de nueva línea) |
| %o | Entero formateado como valor octal |
| %s | String |
| %t | Fecha/Tiempo |
| %x | Entero formateado como valor hexadecimal |

Vamos a usar el método printf() para formatear la salida del método area() de la clase Circle. Por default, el método area() calcula el área del círculo e imprime usando el método println() tradicional. Sin embargo, quieres controlar el formato de salida. Más específicamente quieres imprimir el área con solo dos dígitos decimales de precisión. El siguiente código muestra cómo:

class Circle {

private int x, y, radius;

public Circle(int x, int y, int radius) {

this.x = x;

this.y = y;

this.radius = radius;

}

void area() {

double tempArea = Math.PI \* radius \* radius;

System.out.println(“Circle area using default formatting with println: ” + tempArea)

System.out.printf(“Circle area using format specifier with printf: %.2f”, tempArea);

}

public static void main(String... args) {

Circle circle = new Circle(10,10,5);

circle.area();

}

}

Bueno, puedes ver que la primera línea contiene la salida no formateada y la segunda sí está formateada.

Vamos a usar más opciones de formato en un ejemplo. Supón que tú quieres imprimir una tabla con los jugadores de futbol con sus nombres, partidos jugados, goles y coles por partido. Sin embargo hay unas limitaciones:

* Quieres imprimir el nombre de los jugadores alineados a la izquierda
* Quieres especificar al menos 15 caracteres del nombre de los jugadores
* Quieres imprimir cada columna a una distancia de tab-stop
* Quieres especificar solo un punto de precisión en goles promedio

Veamos el siguiente código:

public class FormattedTable {

static void line() {

System.out.println(“---------------------------------------------------------------”);

}

static void printHeader() {

System.out.printf(“%-15s \t %s \t %s \t %s \n”,“Player”,“Matchers”,“Goals”,“av G”);

}

static void printRow(String palyer, int matches, int goals) {

System.out.printf(“%-15s \t %10d \t\t %d \t\t %.1f \n”,player,matches,goals, ((float)goals/(float)matches));

}

public static void main(String... args) {

FormattedTable.line();

FormattedTable.printHeader();

FormattedTable.line();

FormattedTable.printRow(“Demando”, 100, 122);

FormattedTable.printRow(“Mushi”, 80, 100); // etc

}

}

Vamos a analizar el formato definido en el método printRow(). La primera parte del formato es “%-15s”. Aquí, la expresión empieza con %, que indica el comienzo de un especificador de formato. El siguiente símbolo es “-” que es usado para que la cadena se alinee a la izquierda. El número “15” especifica el ancho de la cadena y finalmente el tipo de dato con String con la letra “s” indica que el dato de entrada es un String. El siguiente especificador de formato es “%5d”, que significa que se espera un entero que será mostrado con cinco dígitos como mínimo. El último especificador de formato es “%.1f”, que espera un número de punto flotante que será mostrado con un dígito decimal de precisión. Todos los especificadores de formato son separados con uno o más “\t” (tab-stop) para hacer espacio entre las columnas.

### Puntos a recordar

* Si no especificas ningún especificador de formato, el método printf() no imprimirá nada del argumento dado
* Las flags como “-”, “^” o “0” tienen sentido solo cuando especificas (ancho) width con el especificador de formato
* Puedes imprimir incluso el carácter “%” en la cadena formatead, sin embargo necesitas usar una secuencia de escape para esto. En el especificador de formato, % es un carácter de escape, lo que significa que necesitas usar “%%” para imprimir un único “%”
* Si tú no provees el tipo de dato especificado por el especificador de formato, puedes obtener una excepción IllegalFormatConversionException. Por ejemplo, si provees un string en lugar de un entero obtienes esta excepción
* Si quieres dar formato a la cadena primero y después imprimirla puedes usar el método estático en la clase String format(). Veamos el ejemplo del método printRow() re-implementado con usando el método format()

void printRow(String player, int matches, int goals) {

String str = String.format(“%-15s \t %5d \t\t %d \t\t %.1f \n”,player,matches,goals, ((float)goals/(float)matches));

System.out.print(str);

}

NOTA: En las regex el punto “.” cuando no tiene carácter de escape “\” funciona como un comodín de un carácter.

NOTA: no puedes usar el regex “\b” cuando al principio o al final cuando los caracteres de la cadena no son caracteres de palabras (Letras, números o guion bajo)

#### Resumen

Búsqueda, análisis y construcción de Strings

* Puedes usar las versiones sobrecargadas del método indexOf() en la clase String para buscar hacia adelante en una cadena, lastIndexOf() para buscar al final y regionMatches() para comparar una región de texto en una cadena
* Para convertir de tipos primitivos a String, puedes hacer uso del método sobrecargado valueOf(), que toma un tipo primitivo como argumento y retorna su String. Para convertir de un String a un tipo primitivo, puedes hacer uso de los métodos parse disponibles en la clase envoltorio correspondiente.
* Para analizar una cadena puedes usar el método split() disponible en la clase String, el cual toma un delimitador como argumento y este es una expresión regular.

Expresiones regulares (regex)

* Una regex define un patrón de búsqueda que puede ser utilizado para ejecutar operaciones como búsqueda y manipulación de Strings
* Usa las clases Pattern y Matcher siempre que ejecutes búsquedas y reemplazos de cadenas, ya que estas clases son más eficientes y rápidas que cualquier otra forma
* Puedes formar grupos con las regex. Estos grupos pueden ser usados para especificar cuantificadores en un sub-juego de la regex. Estos grupos pueden ser usados como referencia trasera

Formato de strings

* El método printf() y el método format() en la clase String usa banderas de formato para formatear Strings
* cada especificador de formato empieza con el símbolo % seguido de flag, ancho y precisión; y termina con el especificador del tipo de dato esperado. En esta cadena, las flags, ancho y precisión son opcionales, pero el signo % y el tipo de dato esperado son obligatorios