Arrays / Matrices / String

Matrices Java

Las matrices se utilizan para almacenar múltiples valores en una sola variable, en lugar de declarar variables separadas para cada valor.

Para declarar una matriz, defina el tipo de variable entre **corchetes** :

```
String[] cars;
```

Ahora hemos declarado una variable que contiene una serie de cadenas. Para insertarle valores, puede colocar los valores en una lista separada por comas, dentro de llaves:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

Para crear una matriz de números enteros, puedes escribir:

```
int[] myNum = {10, 20, 30, 40};
```

Acceder a los elementos de una matriz

Puede acceder a un elemento de matriz consultando el número de índice.

Esta declaración accede al valor del primer elemento en los automóviles:

Ejemplo

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo
cars[0] = "Opel";
```

Ejemplo

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
// Now outputs Opel instead of Volvo
```

Longitud de la matriz

Para saber cuántos elementos tiene una matriz, use la `length`propiedad:

Ejemplo

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
```

Matrices

Una matriz es una estructura de datos que almacena elementos del mismo tipo

Declarar una matriz, sintaxis de Java	**Ejemplos**	**Comentario**

1.	`tipo de datos [] nombre de matriz;`	int[] myArray; Object[] arrayOfObjects;	Es aconsejable declarar una matriz de esta manera

Ejemplo

```
String[] seasons = new String {"Winter", "Spring", "Summer",
"Autumn"};
for (int i = 0; i < 4; i++) {
System.out.println(seasons[i]);
}
```

El programa mostrará lo siguiente:

```
Winter
Spring
Summer
Autumn
```

ahora un ejemplo de un programa que suma , resta matriz identidad y multiplica matrices

explicación

este ejemplo utilizamos matrices cuadradas aleatorias para que sea mas rapido pero podríamos utilizar matrices introducidas por el usuario(en la función rellenar dejo comentado como sería para introducir los datos por el usuario).

La suma y resta de matrices tiene la condición de que las matrices tienen que ser de la misma dimensión, tienen que tener el mismo número de filas y de columnas.
Para efectuar la suma sumamos elemento a elemento:
el elemento (0,0) de la matriz 1 lo sumamos con el elemento (0,0) de la matriz 2 y obtenemos el elemento (0,0) de la matriz suma,
el elemento (0,1) de la matriz 1 lo sumamos con el elemento (0,1) de la matriz 2 y obtenemos el elemento (0,1) de la matriz suma y así sucesivamente.
Para restar seguimos el procedimiento anterior pero cambiando suma por resta.

Para multiplicar matrices las matrices tienen que cumplir la condición de que:
las filas de la matriz 1 = las columnas de la matriz 2
las columnas de la matriz 1 = las filas matriz 2
La matriz resultante tendría la dimensión :
[filas matriz 1 , columnas matriz 2]
Para efectuar la multiplicación efectuamos la operacion:
matrizproducto(0,0) = matriz1(0,**0**) * matriz2(**0**,0) + matriz1(0,**1**) * matriz2(**1**,0)+matriz1(0,**2**)*matriz2(**2**,0)+matriz1(0,**3**)*matriz2(**3**,0)
Con esto obtendríamos el elemento (0,0) de la matriz producto.
matrizproducto(0,1) = matriz1(0,**0**) * matriz2(**0**,1) + matriz1(0,**1**) * matriz2(**1**,1)+matriz1(0,**2**)*matriz2(**2**,1)+matriz1(0,**3**)*matriz2(**3**,1)
Y asi sucesivamente hasta hayar todos los elementos de la matriz producto

(vete a la practica de matrices para ver el codigo)

Expresiones regulares

¿Qué es una expresión regular (regex)?

De hecho, una expresión regular es un patrón para encontrar una cadena en el texto. En Java, la representación original de este patrón es siempre una cadena, es decir, un objeto de la `String`clase. Sin embargo, no es cualquier cadena que se pueda compilar en una expresión regular, solo cadenas que se ajustan a las reglas para crear expresiones regulares. La sintaxis se define en la especificación del lenguaje. Las expresiones regulares se escriben con letras y números, así como con metacaracteres, que son caracteres que tienen un significado especial en la sintaxis de las expresiones regulares. Por ejemplo:

```
String regex = "java"; // The pattern is "java";
String regex = "\\d{3}"; // The pattern is three
digits;
```

Crear expresiones regulares en Java

La creación de una expresión regular en Java implica dos pasos simples:

1. escríbalo como una cadena que cumpla con la sintaxis de expresiones regulares;
2. compilar la cadena en una expresión regular;

En cualquier programa Java, empezamos a trabajar con expresiones regulares creando un `Pattern`objeto. Para hacer esto, necesitamos llamar a uno de los dos métodos estáticos de la clase: `compile`. El primer método toma un argumento: una cadena literal que contiene la expresión regular, mientras que el segundo toma un argumento adicional que determina la configuración de coincidencia de patrones:

```
public static Pattern compile (String literal)
public static Pattern compile (String literal, int
flags)
```

La lista de valores potenciales del `flags`parámetro se define en `Pattern`clase y está disponible para nosotros como variables de clase estáticas. Por ejemplo:

```
Pattern pattern = Pattern.compile("java",
Pattern.CASE_INSENSITIVE); // Pattern-matching will be
case insensitive.
```

Básicamente, la `Pattern`clase es un constructor de expresiones regulares. Bajo el capó, el `compile`método llama al `Pattern`constructor privado de la clase para crear una representación compilada. Este mecanismo de creación de objetos se implementa de esta manera para crear objetos inmutables. Cuando se crea una expresión regular, se comprueba su sintaxis. Si la cadena contiene errores, se `PatternSyntaxException`genera un.

Sintaxis de expresiones regulares

La sintaxis de las expresiones regulares se basa en los `<([{\^-=$!|]})?*+.>`caracteres, que se pueden combinar con letras. Dependiendo de su función, se pueden dividir en varios grupos:

1. Metacaracteres para hacer coincidir los límites de líneas o texto

Metacarácter	Descripción
^	principio de una linea
ps	final de una línea
\b	límite de palabras

\B	límite de no palabra
\A	comienzo de la entrada
\GRAMO	final del partido anterior
\Z	final de la entrada

\z	final de la entrada

2. Metacaracteres para emparejar clases de caracteres predefinidas

Metacarácter	**Descripción**

\d	dígito
\D	sin dígitos
\s	carácter de espacio en blanco
\S	carácter sin espacio en blanco

\w	carácter alfanumérico o guión bajo
\W	cualquier carácter excepto letras, números y guiones bajos
.	cualquier personaje

3. Metacaracteres para emparejar caracteres de control

Metacarácter	Descripción
\t	carácter de tabulación
\norte	carácter de nueva línea
\r	retorno de carro

\F	carácter de avance de línea
\u0085	carácter de la siguiente línea
\u2028	separador de linea
\u2029	separador de párrafo

4. Metacaracteres para clases de personajes

Metacarácter	**Descripción**
[a B C]	cualquiera de los caracteres enumerados (a, b o c)
[^ abc]	cualquier carácter distinto de los enumerados (no a, b, o c)
[a-zA-Z]	rangos combinados (caracteres latinos de la a a la z, sin distinción entre mayúsculas y minúsculas)

[anunc io[mp]]	unión de caracteres (de la a a la d y de la m a la p)
[az&&[def]]	intersección de caracteres (d, e, f)
[az&&[^bc]]	resta de caracteres (a, dz)

5. Metacaracteres para indicar el número de caracteres (cuantificadores). Un cuantificador siempre va precedido de un carácter o grupo de caracteres.

Metacarácter	Descripción
?	uno o ninguno
*	cero o más veces
+	una o más veces

{norte}	n veces
{norte,}	n o más veces
{Nuevo Méjico}	al menos n veces y no más de m veces

Cuantificadores codiciosos

Una cosa que debe saber sobre los cuantificadores es que vienen en tres variedades diferentes: codiciosos, posesivos y reacios. Un cuantificador se convierte en posesivo agregando un `+`carácter " " después del cuantificador. Lo haces reacio agregando " `?`". Por ejemplo:

```
"A.+a" // greedy
"A.++a" // possessive
"A.+?a" // reluctant
```

Intentemos usar este patrón para comprender cómo funcionan los diferentes tipos de cuantificadores. Por defecto, los cuantificadores son codiciosos. Esto significa que buscan la coincidencia más larga de la cadena. Si ejecutamos el siguiente código:

```
public static void main(String[] args) {
    String text = "Fred Anna Alexander";
    Pattern pattern = Pattern.compile("A.+a");
    Matcher matcher = pattern.matcher(text);
    while (matcher.find()) {

System.out.println(text.substring(matcher.start(),
matcher.end()));
    }
}
```

obtenemos esta salida:

```
Anna Alexa
```

Caracteres de escape en expresiones regulares

Debido a que una expresión regular en Java, o mejor dicho, su representación original, es un literal de cadena, debemos tener en cuenta las reglas de Java con respecto a los literales de cadena. En particular, el carácter de barra invertida " \" en los literales de cadena en el código fuente de Java se interpreta como un carácter de control que le dice al compilador que el siguiente carácter es especial y debe interpretarse de una manera especial. Por ejemplo:

```
String s = "The root directory is \nWindows"; // Move
"Windows" to a new line
String s = "The root directory is \u00A7Windows"; //

Insert a paragraph symbol before "Windows"
```

Esto significa que los literales de cadena que describen expresiones regulares y utilizan `\`caracteres " " (es decir, para indicar metacaracteres) deben repetir las barras invertidas para asegurarse de que el compilador de bytecode de Java no malinterprete la cadena. Por ejemplo:

```
String regex = "\\s"; // Pattern for matching a whitespace character
String regex = "\"Windows\"";  // Pattern for matching "Windows"
```

Las barras invertidas dobles también se deben usar para escapar de los caracteres especiales que queremos usar como caracteres "normales". Por ejemplo:

```
String regex = "How\\?";  // Pattern for matching "How?"
```

Métodos de la clase Pattern

La `Pattern`clase tiene otros métodos para trabajar con expresiones regulares:

- `String pattern()` – devuelve la representación de cadena original de la expresión regular utilizada para crear el `Pattern`objeto:

```
Pattern pattern = Pattern.compile("abc");
```

- `System.out.println(pattern.pattern()); // "abc"`
- `static boolean matches(String regex, CharSequence input)` – le permite comparar la expresión regular pasada como regex con el texto pasado como `input`. Devoluciones: verdadero: si el texto coincide con el patrón;

falso – si no es así;

Por ejemplo:

```
System.out.println(Pattern.matches("A.+a","Anna")); //
true
```

- `System.out.println(Pattern.matches("A.+a","Fred Anna Alexander")); // false`
- `int flags()` – devuelve el valor del `flags`conjunto de parámetros del patrón cuando se creó el patrón o 0 si el parámetro no se configuró. Por ejemplo:

```
Pattern pattern = Pattern.compile("abc");
System.out.println(pattern.flags()); // 0
Pattern pattern =
Pattern.compile("abc",Pattern.CASE_INSENSITIVE);
```

- `System.out.println(pattern.flags()); // 2`
- `String[] split(CharSequence text, int limit)` – divide el texto pasado en una `String`matriz. El `limit`parámetro indica el número máximo de coincidencias buscadas en el texto:
  - si `limit > 0`- `limit-1`coincide;
  - if `limit < 0`- todas las coincidencias en el texto
  - si `limit = 0`– todas las coincidencias en el texto, las cadenas vacías al final de la matriz se descartan;
- Por ejemplo:

```
public static void main(String[] args) {
    String text = "Fred Anna Alexa";
    Pattern pattern = Pattern.compile("\\s");
    String[] strings = pattern.split(text,2);
    for (String s : strings) {
        System.out.println(s);
```

```
    }
    System.out.println("---------");
    String[] strings1 = pattern.split(text);
    for (String s : strings1) {
        System.out.println(s);
    }
```

- }
- Salida de la consola:

```
Fred
Anna Alexa
---------
Fred
Anna
```

- `Alexa`
- A continuación, consideraremos otro de los métodos de la clase utilizados para crear un `Matcher`objeto.

Métodos de la clase Matcher

Las instancias de la `Matcher`clase se crean para realizar coincidencias de patrones. `Matcher`es el "motor de búsqueda" de expresiones regulares. Para realizar una búsqueda, necesitamos darle dos cosas: un patrón y un índice de inicio. Para crear un `Matcher`objeto, la `Pattern`clase proporciona el siguiente método: `public Matcher matcher(CharSequence input)` El método toma una secuencia de caracteres, que se buscará. Esta es una instancia de una clase que implementa la `CharSequence`interfaz. Puede pasar no solo un `String`, sino también un `StringBuffer`, `StringBuilder`, `Segment`o `CharBuffer`. El patrón es un `Pattern`objeto en el que `matcher`se llama al método. Ejemplo de creación de un emparejador:

```
Pattern p = Pattern.compile("a*b"); // Create a
compiled representation of the regular expression
Matcher m = p.matcher("aaaaab"); // Create a "search
engine" to search the text "aaaaab" for the pattern
"a*b"
```

Ahora podemos usar nuestro "motor de búsqueda" para buscar coincidencias, obtener la posición de una coincidencia en el texto y reemplazar el texto usando los métodos de la clase. El `boolean find()` método busca la siguiente coincidencia en el texto. Podemos usar este método y una declaración de bucle para analizar un texto completo como parte de un modelo de evento. En otras palabras, podemos realizar las operaciones necesarias cuando ocurre un evento, es decir, cuando encontramos una coincidencia en el texto. Por ejemplo, podemos usar los métodos `int start()` y de esta clase `int end()` para determinar la posición de una coincidencia en el texto. Y podemos usar los métodos `String replaceFirst(String replacement)` y `String replaceAll(String replacement)` para reemplazar coincidencias con el valor del parámetro de reemplazo. Por ejemplo:

```
public static void main(String[] args) {
    String text = "Fred Anna Alexa";
    Pattern pattern = Pattern.compile("A.+?a");

    Matcher matcher = pattern.matcher(text);
    while (matcher.find()) {
        int start=matcher.start();
        int end=matcher.end();
        System.out.println("Match found: " +
text.substring(start, end) + " from index "+ start + "
through " + (end-1));
    }
    System.out.println(matcher.replaceFirst("Ira"));
    System.out.println(matcher.replaceAll("Mary"));
    System.out.println(text);
}
```

Producción:

```
Match found: Anna from index 5 through 8
Match found: Alexa from index 10 through 14
Fred Ira Alexa
Fred Mary Mary
Fred Anna Alexa
```

Objetos , Herencia Y Polimorfismo

(voy a subir varios archivos pero herencia usare el de zoo de la profe)

Lectura de Fichero (todo)

(referencia tanto videojuego practica 45- 46 como nulidades)

XML

(referencia practica 48)