Core Java APIs

1. Creating and Manipulating Strings

Una cadena es básicamente una secuencia de caracteres. Por ejemplo:

```
String name = "Fluffy";
String name = new String("Fluffy");
```

Ellos son sutilmente diferentes. Por ahora, solo recuerda que la clase `String` es especial y no necesita ser instanciada con `new`.

1.1. CONCATENATION

Las reglas principales de concatenación:

1. Si ambos operandos son numéricos, + significa suma numérica.
2. Si cualquier operando es un `String`, + significa concatenación.
3. La expresión se evalúa de izquierda a derecha.

Por ejemplo:

```
System.out.println(1 + 2);            // 3
System.out.println("a" + "b");        // ab
System.out.println("a" + "b" + 3 + 2);   // ab32
System.out.println(1 + 2 + "c");      // 3c
```

Otro ejemplo:

```
int three = 3;
String four = "4";
System.out.println(1 + 2 + three + four);
```

La salida es:

```
64
```

Sólo tienes que recordar que hace +=. Entonces, `s += "2"` significa lo mismo que `s = s + "2"`. Por ejemplo:

```
4: String s = "1";             // s currently holds "1"
5: s += "2";                   // s currently holds "12"
6: s += 3;                     // s currently holds "123"
7: System.out.println(s);      // 123
```

1.2. IMMUTABILITY

Una vez que se crea un objeto `String`, no se puede cambiar. No puede hacerse más grande o más pequeño, y no puede cambiar uno de los caracteres dentro de él.

Mutable es otra palabra para cambiable. Inmutable es lo opuesto. `String` es inmutable.

MORE ON IMMUTABILITY
Considere el siguiente código:

```
class Mutable {
```

```
   private String s;
   public void setS(String newS){ s = newS; }  // Setter makes it mutable
   public String getS() { return s; }
}
final class Immutable {
   private String s = "name";
   public String getS() { return s; }
}
```

`Immutable` solo tiene un `getter`. No hay forma de cambiar el valor de `s` una vez configurado. `Mutable` tiene un setter. Esto permite que la referencia a `s` cambie para apuntar a un `String` diferente más adelante. Tenga en cuenta que, aunque la clase `String` es inmutable, aún se puede usar en una clase mutable. Incluso puede hacer que la variable de instancia sea `final` para que el compilador le recuerde si accidentalmente cambia s.

1.3. IMPORTANT STRING METHODS

Para todos estos métodos, debe recordar que una cadena es una secuencia de caracteres y Java cuenta desde 0 cuando se indexa.

1.3.1. length()

El método `length()` devuelve la cantidad de caracteres en la Cadena. La firma del método es la siguiente:

```
int length()
```

Por ejemplo:

```
String string = "animals";
System.out.println(string.length());  // 7
```

1.3.2. charAt()

El método `charAt()` permite consultar a la cadena para encontrar qué carácter tiene un índice específico. La firma del método es la siguiente:

```
char charAt(int index)
```

Por ejemplo:

```
//               0123456
String string = "animals";
System.out.println(string.charAt(0));  // a
System.out.println(string.charAt(6));  // s
System.out.println(string.charAt(7));  // throws exception
                                       // StringIndexOutOfBoundsException:
                                       //String index out of range: 7
```

1.3.3. indexOf()

El método `indexOf()` busca en los caracteres en la cadena y encuentra el primer índice que coincide con el valor deseado. Las firmas de método son las siguientes:

```
int indexOf(char ch)
int indexOf(char ch, int fromIndex)
int indexOf(String str)
int indexOf(String str, int fromIndex)
```

Por ejemplo:
```
String string = "animals";
System.out.println(string.indexOf('a'));          // 0
System.out.println(string.indexOf('a', 4));       // 4
System.out.println(string.indexOf("al"));         // 4
System.out.println(string.indexOf("al", 5));      // -1
```

1.3.4. substring()

El método `substring()` devuelve partes de la cadena. Las firmas de método son las siguientes:

```
int substring(int beginIndex)
int substring(int beginIndex, int endIndex)
```

Por ejemplo:
```
String string = "animals";
System.out.println(string.substring(3));  // mals
System.out.println(string.substring(string.indexOf('m'))); // mals
System.out.println(string.substring(3, 4)); // m
System.out.println(string.substring(3, 7)); // mals
```

Los siguientes ejemplos no son tan obvios:
```
System.out.println(string.substring(3, 3)); // empty string
System.out.println(string.substring(3, 2));  // throws exception
System.out.println(string.substring(3, 8)); // throws exception
```

1.3.5. toLowerCase() and toUpperCase()

Cambia a mayúsculas o minúsculas de acuerdo con el método invocado, afectando solo los caracteres y dejando intacto el resto. Las firmas de método son las siguientes:

```
String toLowerCase(String str)
String toUpperCase(String str)
```

Por ejemplo:
```
String string = "animals";
System.out.println(string.toUpperCase());  // ANIMALS
System.out.println("Abc123".toLowerCase());  // abc123
```

1.3.6. equals() and equalsIgnoreCase()

El método `equals()` comprueba si dos objetos `String` contienen exactamente los mismos caracteres en el mismo orden. El método `equalsIgnoreCase()` verifica si dos objetos `String` contienen los mismos caracteres, con la excepción de que convertirá a mayúsculas o minúsculas los caracteres si es necesario. Las firmas de método son las siguientes:

```
boolean equals(String str)
boolean equalsIgnoreCase(String str)
```

Por ejemplo:
```
System.out.println("abc".equals("ABC"));  // false
System.out.println("ABC".equals("ABC"));  // true
```

```
System.out.println("abc".equalsIgnoreCase("ABC"));  // true
```

1.3.7. startsWith() and endsWith()

Los métodos `startsWith()` y `endsWith()` analizan si el valor proporcionado coincide con una parte de la Cadena. Las firmas de método son las siguientes:

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)
```

Por ejemplo:

```
System.out.println("abc".startsWith("a")); // true
System.out.println("abc".startsWith("A")); // false
System.out.println("abc".endsWith("c")); // true
System.out.println("abc".endsWith("a")); // false
```

1.3.8. replace()

El método `replace()` hace una búsqueda simple y lo reemplaza en la cadena. Las firmas de método son las siguientes:

```
String replace(char oldChar, char newChar)
String replace(CharSequence oldChar, CharSequence newChar)
```

Por ejemplo:

```
System.out.println("abcabc".replace('a', 'A')); // AbcAbc
System.out.println("abcabc".replace("a", "A")); // AbcAbc
```

1.3.9. contains()

El método `contains()` también busca coincidencias en `String`. La firma del método es la siguiente:

```
boolean contains(String str)
```

Por ejemplo:

```
System.out.println("abc".contains("b")); // true
System.out.println("abc".contains("B")); // false
```

1.3.10. trim(), strip(), stripLeading(), and stripTrailing()

El método `trim()` elimina espacios en blanco ubicados en el principio y el final de una cadena. En términos del examen, el espacio en blanco consiste en espacios junto con los caracteres `\t` (tab) y `\n` (nueva línea). Otros caracteres, como `\r` (retorno de carro), también se incluyen en lo que se recorta.

El método `strip()` es nuevo en Java 11. Hace todo lo que hace `trim()`, pero es compatible con Unicode. Además, los métodos `stripLeading()` y `stripTrailing()` se agregaron en Java 11. El método `stripLeading()` elimina los espacios en blanco del comienzo de `String` y deja los del final. El método `stripTrailing()` hace lo contrario. Elimina los espacios en blanco del final de la Cadena y deja los del principio.

La firma del método es la siguiente:

```
String strip()
String stripLeading()
String stripTrailing()
String trim()
```

Por ejemplo:

```
System.out.println("abc".strip());                 // abc
System.out.println("\t   a b c\n".strip());        // a b c

String text = " abc\t ";
System.out.println(text.trim().length());          // 3
System.out.println(text.strip().length());         // 3
System.out.println(text.stripLeading().length()); // 5
System.out.println(text.stripTrailing().length());// 4
```

1.3.11. intern()

El método `intern()` devuelve el valor del *pool* de `String` sí está allí. De lo contrario, agrega el valor al grupo de cadenas. Esto se verá más adelante. La firma del método es la siguiente:

```
String intern()
```

1.4. METHOD CHAINING

El método de encadenamiento se puede ver del siguiente modo:

```
String start = "AniMaL   ";
String trimmed = start.trim();                 // "AniMaL"
String lowercase = trimmed.toLowerCase();      // "animal"
String result = lowercase.replace('a', 'A');   // "AnimAl"
System.out.println(result);
```

El código anterior puede ser escrito de forma equivalente del siguiente modo:

```
String result = "AniMaL   ".trim().toLowerCase().replace('a', 'A');
System.out.println(result);
```

¿Cuál crees que es el resultado de este código?

```
5: String a = "abc";
6: String b = a.toUpperCase();
7: b = b.replace("B", "2").replace('C', '3');
8: System.out.println("a=" + a);
9: System.out.println("b=" + b);
```

Obtenemos:

```
a=abc
b=A23
```

2. Using the StringBuilder Class

2.1. MUTABILITY AND CHAINING

Cuando encadenamos las llamadas al método `String`, el resultado fue un nuevo `String` con la respuesta. Encadenar objetos `StringBuilder` no funciona de esta manera. En cambio, `StringBuilder` cambia su propio estado y devuelve una referencia a sí mismo. Por ejemplo:

```
4: StringBuilder a = new StringBuilder("abc");
5: StringBuilder b = a.append("de");
6: b = b.append("f").append("g");
7: System.out.println("a=" + a);
8: System.out.println("b=" + b);
```

¿Dijo que ambos imprimen `"abcdefg"`? Bien. Aquí solo hay un objeto `StringBuilder`.

2.2. CREATING A STRINGBUILDER

Hay tres formas de construir un `StringBuilder`:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);        //capacity = 10
```

Size vs. Capacity
El `size` es el número de caracteres actualmente en la secuencia, y `capacity` es el número de caracteres que la secuencia puede contener actualmente. Por ejemplo:

```
StringBuilder sb = new StringBuilder(5); // size=0, capacity=5
```

0	1	2	3	4

```
sb.append("anim");                       // size=4, capacity=5
```

a	n	i	m	
0	1	2	3	4

```
sb.append("als");                        // size=7, capacity>7 (hay incremento)
```

a	n	i	m	a	l	s		
0	1	2	3	4	5	6	7	…

2.3. IMPORTANT STRINGBUILDER METHODS

2.3.1. charAt(), indexOf(), length(), and substring()

Estos cuatro métodos funcionan exactamente igual que en la clase `String`. Por ejemplo:

```
StringBuilder sb = new StringBuilder("animals");
String sub = sb.substring(sb.indexOf("a"), sb.indexOf("al"));
int len = sb.length();
char ch = sb.charAt(6);
System.out.println(sub + " " + len + " " + ch);
```

Obtenemos:

```
anim 7 s
```

2.3.2. append()

Agrega el parámetro a `StringBuilder` y devuelve una referencia al `StringBuilder` actual. Una de las firmas de método es la siguiente:

```
StringBuilder append(String str)
```

Por ejemplo:

```
StringBuilder sb = new StringBuilder().append(1).append('c');
sb.append("-").append(true);
System.out.println(sb);      // 1c-true
```

2.3.3. insert()

El método `insert()` agrega caracteres a `StringBuilder` en el índice solicitado y devuelve una referencia al `StringBuilder` actual. Al igual que `append()`, hay muchas firmas de métodos para diferentes tipos. Aquí hay uno:

```
StringBuilder insert(int offset, String str)
```

Por ejemplo:

```
3: StringBuilder sb = new StringBuilder("animals");
4: sb.insert(7, "-");                  // sb = animals-     Similar a append
5: sb.insert(0, "-");                  // sb = -animals-
6: sb.insert(4, "-");                  // sb = -ani-mals
7: System.out.println(sb);
```

2.3.4. delete() and deleteCharAt()

Elimina caracteres de la secuencia y devuelve una referencia al `StringBuilder` actual. Las firmas de método son las siguientes:

```
StringBuilder delete(int start, int end)
StringBuilder deleteCharAt(int index)
```

Por ejemplo:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 3);                  // sb = adef
sb.deleteCharAt(5);                   // throws an exception
```

2.3.5. replace()

El método `replace()` funciona de manera diferente para `StringBuilder` que para `String`. La firma del método es la siguiente:

```
StringBuilder replace(int startIndex, int endIndex, String newString)
```

Por ejemplo:

```
StringBuilder builder = new StringBuilder("pigeon dirty");
```

```
builder.replace(3, 6, "sty");
System.out.println(builder);  // pigsty dirty
```

2.3.6. reverse()

Invierte los caracteres en las secuencias y devuelve una referencia al `StringBuilder` actual. La firma del método es la siguiente:

```
StringBuilder reverse()
```

Por ejemplo:

```
StringBuilder sb = new StringBuilder("ABC");
sb.reverse();
System.out.println(sb);
// La salida ees CBA
```

2.3.7. toString()

El último método convierte un `StringBuilder` en una cadena. La firma del método es la siguiente:

```
String toString()
```

Por ejemplo:

```
String s = sb.toString();
```

3. Understanding Equality

Aprendió a usar == para comparar números y las referencias a objetos se refieren al mismo objeto.

3.1. COMPARING EQUALS() AND ==

Por ejemplo:

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
System.out.println(one == two); // false
System.out.println(one == three); // true
```

La igualdad de cadenas, en parte es más complejo debido a la forma en que la JVM reutiliza los literales de cadenas:

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y);    // true
```

Sin embargo:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x == z); // false
```

Es posible forzar la creación de un nuevo `String`:

```
String x = new String("Hello World");
```

```
String y = "Hello World";
System.out.println(x == y); // false
```

Veamos el siguiente ejemplo:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x.equals(z)); // true
```

Esto funciona porque los autores de la clase `String` implementaron un método estándar `equal()` para verificar los valores dentro de la Cadena en lugar de la Cadena en sí. Si una clase no tiene un método `equals()`, Java determina si las referencias apuntan al mismo objeto, que es exactamente lo que hace ==. En caso de que se lo pregunte, los autores de `StringBuilder` no implementaron `equals()`. Si llama a `equals()` en dos instancias de `StringBuilder`, comprobará la igualdad de referencia.

El examen lo pondrá a prueba en su comprensión de la igualdad con los objetos que definen también. Por ejemplo:

```
1: public class Tiger {
2:   String name;
3:   public static void main(String[] args) {
4:     Tiger t1 = new Tiger();
5:     Tiger t2 = new Tiger();
6:     Tiger t3 = t1;
7:     System.out.println(t1 == t1); // true
8:     System.out.println(t1 == t2); // false
9:     System.out.println(t1.equals(t2)); // false
10:  } }
```

3.2. THE STRING POOL

El `String pool` (Grupo cadenas), también conocido como *pool* interno, es una ubicación en la máquina virtual Java (JVM) que recopila todas estas cadenas. El grupo de cadenas contiene valores literales y constantes que aparecen en su programa. Las cadenas que no están en el grupo de cadenas se recogen como cualquier otro objeto.

Por ejemplo:

```
// Ejemplo 1
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y);    // true

// Ejemplo 2
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x == z); // false

// Ejemplo 3
String singleString = "hello world";
String concat = "hello ";
concat += "world";
System.out.println(singleString == concat);
```

En (1), al ser literales y constantes en tiempo de compilación, se crea una única ubicación en memoria. En (2) y (3) se crean nuevo objetas al llamar a un método u operar sobre estos.

Podemos forzar la creación del Objeto. Por ejemplo:

```
String x = "Hello World";
String y = new String("Hello World");
System.out.println(x == y); // false
```

El primero dice que use el `String pool` normalmente. El segundo dice "No, JVM. Realmente no quiero que uses el `String pool`".

Un último ejemplo:

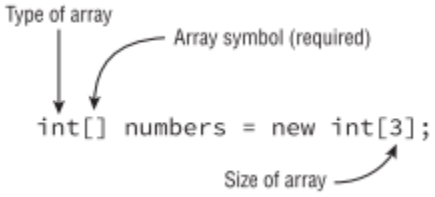
```
15: String first = "rat" + 1;
16: String second = "r" + "a" + "t" + "1";
17: String third = "r" + "a" + "t" + new String("1");
18: System.out.println(first == second);
19: System.out.println(first == second.intern());
20: System.out.println(first == third);
21: System.out.println(first == third.intern());
```

La salida sería:

```
true
true
false
true
```

4. Understanding Java Arrays

4.1. CREATING AN ARRAY OF PRIMITIVES



Al usar este formulario para crear una instancia de una matriz, configure todos los elementos con el valor predeterminado para ese tipo.

Otra forma de crear una matriz es especificar todos los elementos con los que debería comenzar:

```
int[] numbers2 = new int[] {42, 55, 99};
```

As a shortcut, Java lets you write this:

```
int[] numbers2 = {42, 55, 99};
```

Este enfoque se llama una matriz anónima. Es anónimo porque no especifica el tipo y el tamaño. Finalmente, puede escribir el [] antes o después del nombre, y agregar un espacio es opcional. Por ejemplo:

```
int[] numAnimals;
int [] numAnimals2;
int numAnimals3[];
int numAnimals4 [];
```

MULTIPLE "ARRAYS" IN DECLARATIONS
¿Qué tipos de variables de referencia crees que crea el siguiente código?
```
int[] ids, types;
```

La respuesta correcta son dos variables de tipo `int[]`. ¿Qué hay de este ejemplo?
```
int ids[], types;
```

La respuesta es una variable de tipo `int[]` y una variable de tipo `int`.

4.2. CREATING AN ARRAY WITH REFERENCE VARIABLES

Veamos el siguiente ejemplo:
```
public class ArrayType {
  public static void main(String args[]) {
    String [] bugs = { "cricket", "beetle", "ladybug" };
    String [] alias = bugs;
    System.out.println(bugs.equals(alias));     // true
    System.out.println(bugs.toString()); // [Ljava.lang.String;@160bc7c0
} }
```

Devuelve verdadero debido a la igualdad de referencia. El método `equals()` en matrices no mira los elementos de la matriz. Recuerde, esto funcionaría incluso en `int[]` también. `int` es un primitivo; `int[]` es un objeto.

La matriz no asigna espacio para los objetos `String`. En cambio, asigna espacio para una referencia de dónde están realmente almacenados los objetos. Por ejemplo, ¿a qué crees que apunta este conjunto?
```
class Names {
  String names[];
}
```

La respuesta es nula. El código nunca creó una instancia de la matriz, por lo que solo es una variable de referencia nula.

Otro ejemplo, ¿a qué crees que apunta este conjunto?
```
class Names {
  String names[] = new String[2];
}
```

La respuesta es que cada uno de esos dos espacios actualmente es nulo

Finalmente:
```
3: String[] strings = { "stringValue" };
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
6: againStrings[0] = new StringBuilder();   // DOES NOT COMPILE
7: objects[0] = new StringBuilder();        // careful!
                                            // At runtime, the code throws
                                            // an ArrayStoreException
```

4.3. USING AN ARRAY

Intentemos acceder a uno:
```
4: String[] mammals = {"monkey", "chimp", "donkey"};
```

```
5: System.out.println(mammals.length);          // 3
6: System.out.println(mammals[0]);              // monkey
7: System.out.println(mammals[1]);              // chimp
8: System.out.println(mammals[2]);              // donkey
```

Por ejemplo:
```
String[] birds = new String[6];
System.out.println(birds.length);
```

Aunque los 6 elementos de la matriz son nulos, todavía hay 6 de ellos.

Otro ejemplo:
```
5: int[] numbers = new int[10];
6: for (int i = 0; i < numbers.length; i++)
7:   numbers[i] = i + 5;
8: numbers[10] = 3        // Throws ArrayIndexOutOfBoundsException
```

4.4. SORTING

Java facilita ordenar una matriz al proporcionar un método de ordenamiento: `Arrays.sort()`. Por ejemplo:
```
int[] numbers = { 6, 9, 1 };
Arrays.sort(numbers);
for (int i = 0; i < numbers.length; i++)
  System.out.print (numbers[i] +  " ");
```

La salida es:
1 6 9

Inténtalo de nuevo con los tipos de cadenas:
```
String[] strings = { "10", "9", "100" };
Arrays.sort(strings);
for (String string : strings)
  System.out.print(string + " ");
```

Este código muestra 10 100 9. El problema es que `String` se ordena en forma alfabética, y 1 está antes que 9.

4.5. SEARCHING

Java también proporciona una forma conveniente de buscar, pero solo si la matriz ya está ordenada.

Escenario	Resultado
Elemento objetivo encontrado en matriz ordenada	Índice del elemento encontrado
Elemento de destino no es encontrado en la matriz ordenada	Valor negativo que muestra uno más pequeño que el negativo del índice, donde se debe insertar una coincidencia para conservar el orden ordenado
Matriz no ordenada	Una sorpresa: este resultado no es predecible

Por ejemplo:
```
3: int[] numbers = {2,4,6,8};
```

```
4: System.out.println(Arrays.binarySearch(numbers, 2)); // 0
5: System.out.println(Arrays.binarySearch(numbers, 4)); // 1
6: System.out.println(Arrays.binarySearch(numbers, 1)); // -1
7: System.out.println(Arrays.binarySearch(numbers, 3)); // -2
8: System.out.println(Arrays.binarySearch(numbers, 9)); // -5
```

¿Qué crees que pasa en este ejemplo?

```
5: int[] numbers = new int[] {3,2,1};
6: System.out.println(Arrays.binarySearch(numbers, 2));
7: System.out.println(Arrays.binarySearch(numbers, 3));
```

Correctamente resultó 1 como salida. Sin embargo, la línea 7 dio la respuesta incorrecta. Para el examen busque una respuesta sobre resultados impredecibles.

4.6. COMPARING

Java también proporciona métodos para comparar dos arreglos para determinar cuál es "más pequeña".

4.6.1. compare()

Hay un montón de reglas que debe conocer antes de llamar a compare(). Primero debe aprender qué significa el valor de retorno:

1. Un número negativo significa que la primera matriz es más pequeña que la segunda.
2. Un cero significa que los arreglos son iguales.
3. Un número positivo significa que la primera matriz es más grande que la segunda.

Aquí tienes un ejemplo:

```
System.out.println(Arrays.compare(new int[] {1}, new int[] {2}));
```

Veamos las reglas:

1. Si ambos arreglos tienen la misma longitud y los mismos valores en cada lugar en el mismo orden, devuelve cero.
2. Si todos los elementos son iguales pero la segunda matriz tiene elementos adicionales al final, devuelve un número negativo.
3. Si todos los elementos son iguales pero la primera matriz tiene elementos adicionales al final, devuelve un número positivo.
4. Si el primer elemento que difiere es más pequeño en la primera matriz, devuelve un número negativo.
5. Si el primer elemento que difiere es más grande en la primera matriz, devuelve un número positivo.

Algunas reglas para comparaciones especiales:

1. `null` es más pequeño que cualquier otro valor.
2. Para los números, se aplica el orden numérico normal.
3. Para cadenas, una es más pequeña si es un prefijo de otra.
4. Para cadenas/caracteres, los números son más pequeños que las letras.
5. Para cadenas/caracteres, las mayúsculas son más pequeñas que las minúsculas.

Ejemplos:

Primer arreglo	Segundo arreglo	Resultado	Motivo

`new int[] {1, 2}`	`new int[] {1}`	Numero positivo	El primer elemento es el mismo, pero la primera matriz es más larga.
`new int[] {1, 2}`	`new int[] {1, 2}`	Cero	Coincidencia exacta
`new String[] {"a"}`	`new String[] {"aa"}`	Numero negativo	El primer elemento es una sub cadena del segundo.
`new String[] {"a"}`	`new String[] {"A"}`	Numero positivo	Las mayúsculas son más pequeñas que las minúsculas.
`new String[] {"a"}`	`new String[] {null}`	Numero positivo	`null` es más pequeño que una letra.

Finalmente, este código no se compila porque los tipos son diferentes. Al comparar dos matrices, deben ser del mismo tipo de matriz.

```
System.out.println(Arrays.compare(
   new int[] {1}, new String[] {"a"})); // DOES NOT COMPILE
```

4.6.2. mismatch()

Si las matrices son iguales, discordancia () devuelve -1. De lo contrario, devuelve el primer índice donde difieren. Por ejemplo:

```
System.out.println(Arrays.mismatch(new int[] {1}, new int[] {1}));
System.out.println(Arrays.mismatch(new String[] {"a"},new String[] {"A"}));
System.out.println(Arrays.mismatch(new int[] {1, 2}, new int[] {1}));

//La salida seria:
-1
0
1
```

Comparando equals vs. compare vs. mismatch

Método	Cuando los arreglos son iguales	Cuando los arreglos son diferentes
`equals()`	true	false
`compare()`	0	Número positivo o negativo
`mismatch()`	-1	Índice cero o positivo

4.7. VARARGS

Aquí hay tres ejemplos con un método `main()`:

```
public static void main(String[] args)
public static void main(String args[])
public static void main(String... args) // varargs
```

Puede usar una variable definida como `varargs` como si fuera una matriz normal. Por ejemplo, `args.length` y `args[0]` son legales.

4.8. MULTIDIMENSIONAL ARRAYS

4.8.1. Creating a Multidimensional Array

Puede ubicarlos con el tipo o el nombre de la variable en la declaración, como antes:

```
int[][] vars1;           // 2D array
int vars2 [][];          // 2D array
int[] vars3[];           // 2D array
int[] vars4 [], space [][];  // a 2D AND a 3D array
```

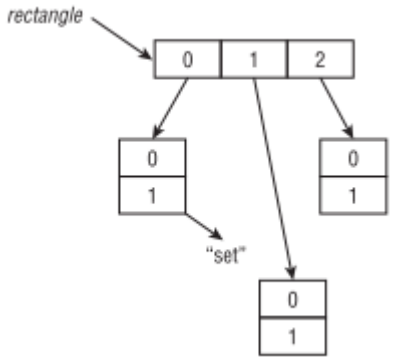
Puede especificar el tamaño de su matriz multidimensional en la declaración si lo desea:

```
String [][] rectangle = new String[3][2];
```

Ahora supongamos que establecemos uno de estos valores:

```
rectangle[0][1] = "set";
```

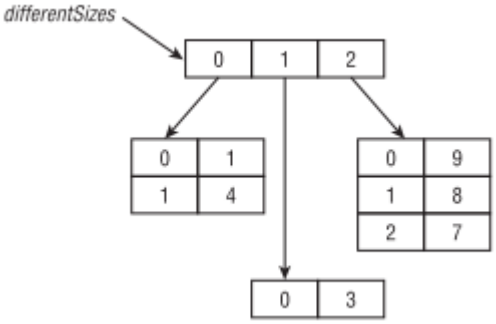
Nos resulta gráficamente:



Si bien esa matriz tiene forma rectangular, no es necesario que sea una matriz. Considere:

```
int[][] differentSize = {{1, 4}, {3}, {9,8,7}};
```

Nos resulta gráficamente:



Otra forma de crear una matriz asimétrica es inicializar solo la primera dimensión de una matriz y definir el tamaño de cada componente de la matriz en una declaración separada:

```
int [][] args = new int[4][];
args[0] = new int[5];
args[1] = new int[3];
```

4.8.2. Using a Multidimensional Array

La operación más común en una matriz multidimensional es recorrerla. Este ejemplo imprime una matriz 2D:

```
int[][] twoD = new int[3][2];
for (int i = 0; i < twoD.length; i++) {
  for (int j = 0; j < twoD[i].length; j++)
    System.out.print(twoD[i][j] + " "); // print element
  System.out.println();                 // time for a new row
}
```

El ejemplo anterior sería más fácil de leer con el bucle for mejorado:

```
for (int[] inner : twoD) {
  for (int num : inner)
    System.out.print(num + " ");
   System.out.println();
}
```

5. Understanding an ArrayList

Al igual que `StringBuilder`, `ArrayList` puede cambiar el tamaño en tiempo de ejecución según sea necesario. Como una matriz, una `ArrayList` es una secuencia ordenada que permite duplicados.

5.1. CREATING AN ARRAYLIST

Hay tres formas de crear una `ArrayList`:

```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);
```

Los ejemplos anteriores eran la antigua forma previa a Java 5 de crear una `ArrayList`. Java 5 introdujo los genéricos, que le permiten especificar el tipo de clase que contendrá `ArrayList`:

```
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>();
```

A partir de Java 7, incluso puede omitir ese tipo desde el lado derecho. Sin embargo, < y > aún se requieren. Esto se llama operador de diamante porque <> se ve como un diamante. Por ejemplo:

```
List<String> list6 = new ArrayList<>();
ArrayList<String> list7 = new List<>(); // DOES NOT COMPILE, List is an interface
```

USING VAR WITH ARRAYLIST

Considere este código mezclando los dos:

```
var strings = new ArrayList<String>();
strings.add("a");
for (String s: strings) {  }
```

El tipo de `var` es `ArrayList <String>`. Esto significa que puede agregar un `String` o recorrer los objetos `String`. ¿Qué pasa si usamos el operador de diamante con `var`?

```
var list = new ArrayList<>();
```

El tipo de `var` es `ArrayList <Object>`.

5.2. USING AN ARRAYLIST

Verá algo nuevo en las firmas de métodos: una "clase" llamada E. E es utilizada por convención en genéricos para significar "cualquier clase que este conjunto pueda contener". Si no especificó un tipo al crear el `ArrayList`, E significa `Object`.

5.2.1. add()

Inserta un nuevo valor en `ArrayList`. Las firmas de método son las siguientes:

```
boolean add(E element)
void add(int index, E element)
```

Por ejemplo:

```
ArrayList list = new ArrayList();
list.add("hawk");          // [hawk]
list.add(Boolean.TRUE);    // [hawk, true]
System.out.println(list);  // [hawk, true]
```

Ahora, usemos los genéricos para decirle al compilador que solo queremos permitir objetos `String` en nuestro `ArrayList`:

```
ArrayList<String> safer = new ArrayList<>();
safer.add("sparrow");
safer.add(Boolean.TRUE);    // DOES NOT COMPILE
```

Ahora intentemos agregar valores múltiples a diferentes posiciones:

```
4: List<String> birds = new ArrayList<>();
5: birds.add("hawk");             // [hawk]
6: birds.add(1, "robin");         // [hawk, robin]
7: birds.add(0, "blue jay");      // [blue jay, hawk, robin]
8: birds.add(1, "cardinal");      // [blue jay, cardinal, hawk, robin]
9: System.out.println(birds);     // [blue jay, cardinal, hawk, robin]
```

5.2.2. remove()

Los métodos `remove()` eliminan el primer valor coincidente en `ArrayList` o eliminan el elemento en un índice especificado. Las firmas de método son las siguientes:

```
boolean remove(Object object)
E remove(int index)
```

Por ejemplo:

```
3: List<String> birds = new ArrayList<>();
4: birds.add("hawk");      // [hawk]
5: birds.add("hawk");      // [hawk, hawk]
6: System.out.println(birds.remove("cardinal")); // prints false
7: System.out.println(birds.remove("hawk")); // prints true
8: System.out.println(birds.remove(0)); // prints hawk
9: System.out.println(birds);      // []
```

5.2.3. set()

El método `set()` cambia uno de los elementos de `ArrayList` sin cambiar el tamaño. La firma del método es la siguiente:

```
E set(int index, E newElement)
```

Por ejemplo:

```
15: List<String> birds = new ArrayList<>();
16: birds.add("hawk");                    // [hawk]
17: System.out.println(birds.size());     // 1
18: birds.set(0, "robin");               // [robin]
19: System.out.println(birds.size());     // 1
20: birds.set(1, "robin");               // IndexOutOfBoundsException
```

5.2.4. isEmpty() and size()

Los métodos `isEmpty()` y `size()` miran cuántas de las ranuras están en uso. Las firmas de método son las siguientes:

```
boolean isEmpty()
int size()
```

Por ejemplo:

```
System.out.println(birds.isEmpty());  // true
System.out.println(birds.size());     // 0
birds.add("hawk");                    // [hawk]
birds.add("hawk");                    // [hawk, hawk]
System.out.println(birds.isEmpty());  // false
System.out.println(birds.size());     // 2
```

5.2.5. clear()

El método `clear()` proporciona una manera fácil de descartar todos los elementos de `ArrayList`. La firma del método es la siguiente:

```
void clear()
```

Por ejemplo:

```
List<String> birds = new ArrayList<>();
birds.add("hawk");                    // [hawk]
birds.add("hawk");                    // [hawk, hawk]
System.out.println(birds.isEmpty());  // false
System.out.println(birds.size());     // 2
birds.clear();                        // []
System.out.println(birds.isEmpty());  // true
System.out.println(birds.size());     // 0
```

5.2.6. contains()

El método `contains()` comprueba si un determinado valor se encuentra en `ArrayList`. La firma del método es la siguiente:

```
boolean contains(Object object)
```

Por ejemplo:
```
List<String> birds = new ArrayList<>();
birds.add("hawk");                          // [hawk]
System.out.println(birds.contains("hawk")); // true
System.out.println(birds.contains("robin")); // false
```

5.2.7. equals()

Finalmente, `ArrayList` tiene una implementación personalizada de `equals()` para que pueda comparar dos listas para ver si contienen los mismos elementos en el mismo orden.

```
boolean equals(Object object)
```

Por ejemplo:
```
31: List<String> one = new ArrayList<>();
32: List<String> two = new ArrayList<>();
33: System.out.println(one.equals(two));    // true
34: one.add("a");                           // [a]
35: System.out.println(one.equals(two));    // false
36: two.add("a");                           // [a]
37: System.out.println(one.equals(two));    // true
38: one.add("b");                           // [a,b]
39: two.add(0, "b");                        // [b,a]
40: System.out.println(one.equals(two));    // false
```

5.3. WRAPPER CLASSES

Cada tipo primitivo tiene una clase *wrapper* (contenedora), que es un tipo de objeto que corresponde al primitivo:

Primitivo	Wrapper class	Ejemplo de Construcción
`boolean`	`Boolean`	`new Boolean(true)`
`byte`	`Byte`	`new Byte((byte) 1)`
`short`	`Short`	`new Short((short) 1)`
`int`	`Integer`	`new Integer(1)`
`long`	`Long`	`new Long(1)`
`float`	`Float`	`new Float(1.0)`
`double`	`Double`	`new Double(1.0)`
`char`	`Character`	`new Character('c')`

Las clases contenedoras también tienen un método que convierte de nuevo a un primitivo y también hay métodos para convertir una cadena en un primitivo o una clase contenedora. Los métodos de análisis, como `parseInt()`, devuelve un primitivo, y el método `valueOf()` devuelve una clase contenedora. Por ejemplo:

```
int primitive = Integer.parseInt("123");
Integer wrapper = Integer.valueOf("123");
int bad1 = Integer.parseInt("a");               // throws NumberFormatException
Integer bad2 = Integer.valueOf("123.45");       // throws NumberFormatException
```

La siguiente tabla muestra los métodos que necesita reconocer para crear un primitivo o objeto de clase contenedor a partir de una cadena

Wrapper class	Convertiendo `String` a primitivo	Convertiendo `String` a wrapper class
`Boolean`	`Boolean.parseBoolean("true");`	`Boolean.valueOf("TRUE");`
`Byte`	`Byte.parseByte("1");`	`Byte.valueOf("2");`
`Short`	`Short.parseShort("1");`	`Short.valueOf("2");`
`Integer`	`Integer.parseInt("1");`	`Integer.valueOf("2");`
`Long`	`Long.parseLong("1");`	`Long.valueOf("2");`
`Float`	`Float.parseFloat("1");`	`Float.valueOf("2.2");`
`Double`	`Double.parseDouble("1");`	`Double.valueOf("2.2");`
`Character`	None	None

WRAPPER CLASSES AND NULL
Cuando presentamos los primitivas numéricos, mencionamos que no se pueden usar para almacenar valores nulos. Una ventaja de una clase contenedora sobre una primitiva es que debido a que es un objeto, se puede usar para almacenar un valor `null`. Por ejemplo, si está almacenando los datos de ubicación de un usuario usando (latitud, longitud), sería una mala idea para almacenar un punto faltante como (0,0) ya que se refiere a una ubicación real frente a la costa de África donde teóricamente podría estar el usuario.

5.4. AUTOBOXING AND UNBOXING

Desde Java 5, puede simplemente escribir el valor primitivo y Java lo convertirá en la clase contenedora relevante para usted. Esto se llama *autoboxing*. Veamos un ejemplo:

```
4: List<Double> weights = new ArrayList<>();
5: weights.add(50.5);               // [50.5]
6: weights.add(new Double(60));     // [50.5, 60.0]
7: weights.remove(50.5);               // [60.0]
8: double first = weights.get(0);     // 60.0
```

¿Si intentas realizar un *unbox* de un nulo?

```
3: List<Integer> heights = new ArrayList<>();
4: heights.add(null);
5: int h = heights.get(0);          // NullPointerException
```

Otro ejemplo:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);    // {1,2}
numbers.remove(1); // Remueve la posicion 1 -> {1}
System.out.println(numbers);
```

En realidad, produce 1 porque ya hay un método `remove()` que toma un parámetro `int`, Java llama a ese método en lugar de *autoboxing*. Si desea eliminar el 1, puede escribir `numbers.remove(new Integer (1))` para forzar el uso de la clase contenedora.

5.5. CONVERTING BETWEEN ARRAY AND LIST

Debe saber cómo realizar conversiones entre una matriz y una `ArrayList`:

```
3: List<String> list = new ArrayList<>();
4: list.add("hawk");
```

```
5: list.add("robin");
6: Object[] objectArray = list.toArray();
7: System.out.println(objectArray.length);     // 2
8: String[] stringArray = list.toArray(new String[0]);
9: System.out.println(stringArray.length);     // 2
```

El único problema es que se establece de forma predeterminada en una matriz de *Objects*. La línea 8 especifica el tipo de matriz y hace lo que realmente queremos. Java creará una nueva matriz del tamaño adecuado para el valor de retorno. Es una lista de tamaño fijo y también se conoce una Lista respaldada (*backed list*) porque la matriz cambia con ella:

```
20: String[] array = { "hawk", "robin" };     // [hawk, robin]
21: List<String> list = Arrays.asList(array); // returns fixed size list
22: System.out.println(list.size());          // 2
23: list.set(1, "test");                      // [hawk, test]
24: array[0] = "new";                         // [new, test]
25: for (String b : array) System.out.print(b + " ");         // new test
26: list.remove(1);              // throws UnsupportedOperation Exception
```

La línea 26 arroja una excepción porque no podemos cambiar el tamaño de la lista.

5.6. USING VARARGS TO CREATE A LIST

El uso de `varargs` te permite crear una lista de una manera genial:

```
List<String> list1 = Arrays.asList("one", "two");
List<String> list2 = List.of("one", "two");
```

Ambos métodos toman `varargs`, que le permiten pasar una matriz o simplemente escribir los valores de cadena. Ambos métodos crean matrices de tamaño fijo. Si más tarde necesitará agregar o eliminar elementos, aún tendrá que crear una `ArrayList` usando el constructor.

Un resumen de conversiones:

	`toArray()`	`Arrays.asList()`	`List.of()`
Tipo de conversión de	`List`	`Array (or varargs)`	`Array (or varargs)`
Tipo creado	`Array`	`List`	`List`
Permitido para eliminar valores del objeto creado	No	No	No
Permitido cambiar valores en el objeto creado	Si	Si	No
El cambio de valores en el objeto creado afecta al original o viceversa.	No	Si	N/A

5.7. SORTING

Ordenar una `ArrayList` es muy similar a ordenar una matriz:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(99);
numbers.add(5);
numbers.add(81);
```

```
Collections.sort(numbers);
System.out.println(numbers); [5, 81, 99]
```

6. Creating Sets and Maps

6.1. INTRODUCING SETS

7. Review Questions

7.1.

7.2.

7.3.

7.4.

7.5.