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 nueva.

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);    // ab3
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 `+ =`. `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.

1.3. 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 que aparecen en su programa. Las cadenas que no están en el grupo de cadenas se recogen como cualquier otro objeto. Por ejemplo:

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

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

1.4. 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.

1.4.1. 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
```

1.4.2. 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:
```
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
```

1.4.3. 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, index fromIndex)
int indexOf(String str)
int indexOf(String str, index 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
```

1.4.4. 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
```

1.4.5. 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
```

1.4.6. 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
```

1.4.7. 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
```

1.4.8. 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
```

1.4.9. 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
```

1.4.10. trim()

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. La firma del método es la siguiente:

```
public String trim()
```

Por ejemplo:
```
System.out.println("abc".trim());           // abc
System.out.println("\t   a b c\n".trim());  // a b c
```

1.5. 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);
```

Size vs. Capacity
El tamaño es el número de caracteres actualmente en la secuencia, y la capacidad 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
```

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-
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. 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
```

2.3.6. 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();
```

2.4. StringBuilder vs. StringBuffer

`StringBuilder` se agregó a Java en Java 5. Si se encuentra con un código anterior, verá `StringBuffer` utilizado para este propósito. `StringBuffer` hace lo mismo, pero más lentamente porque es seguro para subprocesos.

3. Understanding Equality

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

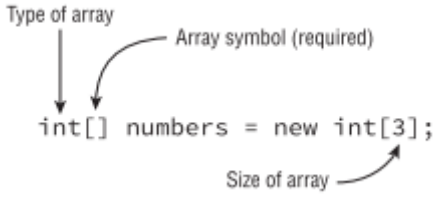
Esto funciona porque los autores de la clase `String` implementaron un método estándar `equal()` para verificar los valores dentro de la Cadena en lugar de la Cadena en sí. Si una clase no tiene un método `equals()`, Java determina si las referencias apuntan al mismo objeto, que es exactamente lo que 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:  } }
```

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. 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.

4.7. Multidimensional Arrays

4.7.1. 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
```

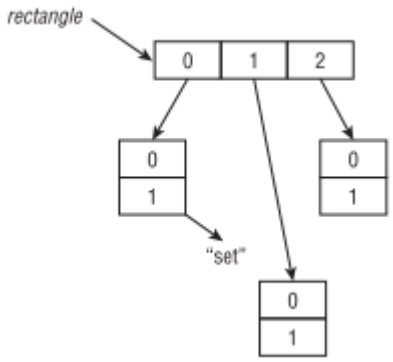
Puede especificar el tamaño de su matriz multidimensional en la declaración si lo desea:

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

Ahora supongamos que establecemos uno de estos valores:

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

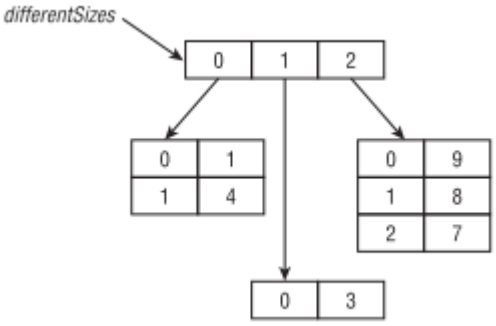
Nos resulta gráficamente:



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];
```

4.7.2. 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
```

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	Converting String to primitive	Converting String to 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

5.4. Autoboxing

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. 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. Working with Dates and Times

En Java 8, Oracle renovó completamente la forma en que trabajamos con fechas y horas. La mayoría de ellos están en el paquete `java.time`.

6.1. Creating Dates and Times

Al trabajar con fechas y horas, lo primero que debe hacer es decidir cuánta información necesita. El examen te da tres opciones:

`LocalDate`: Contiene solo una fecha, sin hora ni zona horaria.
`LocalTime`: Contiene solo una hora, sin fecha y sin zona horaria.
`LocalDateTime`: Contiene fecha y hora, pero no zona horaria.

Oracle recomienda evitar las zonas horarias a menos que realmente las necesite. Si necesita comunicarse a través de zonas horarias, `ZonedDateTime` las maneja.

Por ejemplo:
```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
```

La salida es:
```
2015-01-20
12:45:18.401
2015-01-20T12:45:18.401
```

Espera - Yo no vivo en EE. UU.
El examen no le preguntará acerca de la diferencia entre el 02/03/2015 y el 03/02/2015. Además, Java tiende a usar el formato de 24 horas.

Ambos ejemplos crean la misma fecha:
```
LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);
LocalDate date2 = LocalDate.of(2015, 1, 20);
```

En el ejemplo anterior, ambos pasan en el año, mes y fecha. Aunque es bueno usar las constantes `Month` (para hacer que el código sea más fácil de leer), puede pasar el número `int` del mes directamente. Las firmas de método son las siguientes:
```
public static LocalDate of(int year, int month, int dayOfMonth)
public static LocalDate of(int year, Month month, int dayOfMonth)
```

Nota: Hasta ahora, hemos dicho que Java cuenta comenzando con 0. Bueno, los meses son una excepción. Durante meses en los nuevos métodos de fecha y hora, Java cuenta comenzando desde 1 como lo hacemos los seres humanos.

Al crear una hora, puede elegir qué tan detallado desea ser:
```
LocalTime time1 =  LocalTime.of(6, 15);              // hour and minute
LocalTime time2 =  LocalTime.of(6, 15, 30);         // + seconds
LocalTime time3 =  LocalTime.of(6, 15, 30, 200);    // + nanoseconds
```

Las firmas de método son las siguientes:
```
public static LocalTime of(int hour, int minute)
public static LocalTime of(int hour, int minute, int second)
public static LocalTime of(int hour, int minute, int second, int nanos)
```

Finalmente, podemos combinar fechas y horarios:
```
LocalDateTime dateTime1 = LocalDateTime.of(2015, Month.JANUARY, 20, 6, 15, 30);
LocalDateTime dateTime2 = LocalDateTime.of(date1, time1);
```

Sin embargo, tener tantos números seguidos es difícil de leer. La segunda línea de código muestra cómo puede crear objetos `LocalDate` y `LocalTime` por separado primero y luego combinarlos para crear un objeto `LocalDateTime`. Las firmas de método son las siguientes:

```
public static LocalDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(int year, Month month,
  int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, Month month,
  int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, Month month,
  int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(LocalDate date, LocalTime)
```

Las clases de fecha y hora tienen constructores privados para obligarlo a usar los métodos estáticos:

```
LocalDate d = new LocalDate(); // DOES NOT COMPILE
// java.time.DateTimeException: Invalid value for DayOfMonth
// (valid values 1 - 28/31): 32
```

6.2. Manipulating Dates and Times

Las clases de fecha y hora son inmutables, al igual que `String`:

```
12: LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
13: System.out.println(date);          // 2014-01-20
14: date = date.plusDays(2);
15: System.out.println(date);          // 2014-01-22
16: date = date.plusWeeks(1);
17: System.out.println(date);          // 2014-01-29
18: date = date.plusMonths(1);
19: System.out.println(date);          // 2014-02-28
20: date = date.plusYears(5);
21: System.out.println(date);          // 2019-02-28
```

Esta vez, trabajemos con `LocalDateTime`:

```
22: LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
23: LocalTime time = LocalTime.of(5, 15);
24: LocalDateTime dateTime = LocalDateTime.of(date, time);
25: System.out.println(dateTime);          // 2020-01-20T05:15
26: dateTime = dateTime.minusDays(1);
27: System.out.println(dateTime);          // 2020-01-19T05:15
28: dateTime = dateTime.minusHours(10);
29: System.out.println(dateTime);          // 2020-01-18T19:15
30: dateTime = dateTime.minusSeconds(30);
31: System.out.println(dateTime);          // 2020-01-18T19:14:30
```

Es común que los métodos de fecha y hora estén encadenados. Por ejemplo:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time)
   .minusDays(1).minusHours(10).minusSeconds(30);
```

Methods in `LocalDate`, `LocalTime`, and `LocalDateTime`:

	¿Puede llamar `LocalDate`**?**	**¿Puede llamar** `LocalTime`**?**	**¿Puede llamar** `LocalDateTime`**?**
`plusYears/minusYears`	`Yes`	`No`	`Yes`
`plusMonths/minusMonths`	`Yes`	`No`	`Yes`
`plusWeeks/minusWeeks`	`Yes`	`No`	`Yes`
`plusDays/minusDays`	`Yes`	`No`	`Yes`

plusHours/minusHours	No	Yes	Yes
plusMinutes/minusMinutes	No	Yes	Yes
plusSeconds/minusSeconds	No	Yes	Yes
plusNanos/minusNanos	No	Yes	Yes

Convertir a long
`LocalDate` y `LocalDateTime` tienen un método para convertirlos en equivalentes long en relación con 1970. Eso es lo que UNIX comenzó a usar para los estándares de fecha, por lo que Java lo reutilizó:

- `LocalDate` tiene `toEpochDay()`, que es la cantidad de días desde el 1 de enero de 1970.
- `LocalDateTime` tiene `toEpochTime()`, que es el número de segundos desde el 1 de enero de 1970.
- `LocalTime` no tiene un método de época. Como representa un tiempo que ocurre en cualquier fecha, no tiene sentido compararlo en 1970.

6.3. Working with Periods

Hay cinco formas de crear una clase `Period`:

```
Period annually = Period.ofYears(1);              // every 1 year
Period quarterly = Period.ofMonths(3);            // every 3 months
Period everyThreeWeeks = Period.ofWeeks(3);       // every 3 weeks
Period everyOtherDay = Period.ofDays(2);          // every 2 days
Period everyYearAndAWeek = Period.of(1, 0, 7);    // every year and 7 days
```

No puede encadenar métodos al crear un Período. El siguiente código parece que es equivalente al ejemplo `everyYearAndAWeek`, pero no lo es. Solo se utiliza el último método porque los métodos `Period.ofXXX` son métodos estáticos:

```
Period wrong = Period.ofYears(1).ofWeeks(1);          // every week
```

Es por eso por lo que el método `of()` nos permite pasar el número de años, meses y días. Todos están incluidos en el mismo período. Probablemente ya hayas notado que `Period` es un día o más de tiempo. También hay `Duration`, que está destinada a unidades de tiempo más pequeñas. Para `Duration`, puede especificar el número de días, horas, minutos, segundos o nanosegundos. `Duration` no es parte del examen ya que funciona aproximadamente de la misma manera que `Period`.

Lo último que debe saber sobre `Period` es con qué objetos se puede usar. Veamos un código:

```
3: LocalDate date = LocalDate.of(2015, 1, 20);
4: LocalTime time = LocalTime.of(6, 15);
5: LocalDateTime dateTime = LocalDateTime.of(date, time);
6: Period period = Period.ofMonths(1);
7: System.out.println(date.plus(period));          // 2015-02-20
8: System.out.println(dateTime.plus(period));          // 2015-02-20T06:15
9: System.out.println(time.plus(period));   // UnsupportedTemporalTypeException
```

La línea 9 intenta agregar un mes a un objeto que solo tiene tiempo. Esto no funcionará. Java lanza una excepción y se quejará de que intentamos utilizar una unidad no admitida: Meses.

6.4. Formatting Dates and Times

A diferencia de la clase `LocalDateTime`, `DateTimeFormatter` se puede usar para formatear cualquier tipo de objeto de fecha u hora:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
```

```
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

ISO es un estándar para las fechas. La salida del código anterior se ve así:

```
2020-01-20
11:12:34
2020-01-20T11:12:34
```

Hay algunos formatos predefinidos que son más útiles:

```
DateTimeFormatter shortDateTime =
  DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(shortDateTime.format(dateTime));     // 1/20/20
System.out.println(shortDateTime.format(date));         // 1/20/20
System.out.println(
  shortDateTime.format(time)); // UnsupportedTemporalTypeException
```

La última línea arroja una excepción porque no se puede formatear un tiempo como una fecha. El método `format()` se declara tanto en los objetos del formateador como en los objetos de fecha/hora, lo que le permite hacer referencia a los objetos en cualquier orden. Las siguientes declaraciones imprimen exactamente lo mismo que el código anterior:

```
DateTimeFormatter shortDateTime =
  DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(dateTime.format(shortDateTime));
System.out.println(date.format(shortDateTime));
System.out.println(time.format(shortDateTime));
```

Eche un vistazo a los métodos de formateo localizados legales e ilegales:

DateTimeFormatter f = DateTimeFormatter ._____(FormatStyle.SHORT);	**Llamando f.format(localDate)**	**Llamando f.format(localDateTime)**	**Llamando f.format(localTime)**
ofLocalizedDate	Legal – shows whole object	Legal – shows just date part	Throws runtime exception
ofLocalizedDateTime	Throws runtime exception	Legal – shows whole object	Throws runtime exception
ofLocalizedTime	Throws runtime exception	Legal – shows just time part	Legal – shows whole object

Hay dos formatos predefinidos que pueden aparecer en el examen: SHORT y MEDIUM. Los otros formatos predefinidos implican zonas horarias, que no están en el examen:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
DateTimeFormatter shortF = DateTimeFormatter
   .ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter mediumF = DateTimeFormatter
  .ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(shortF.format(dateTime));     // 1/20/20 11:12 AM
System.out.println(mediumF.format(dateTime));     // Jan 20, 2020 11:12:34 AM
```

Si no quiere usar uno de los formatos predefinidos, puede crear uno propio. Por ejemplo, este código muestra el nombre del mes:

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
System.out.println(dateTime.format(f));      // January 20, 2020, 11:12
```

Lo máximo que tendrá que hacer es reconocer las piezas de fecha y hora:
MMMM: M representa el mes. Por ejemplo, M muestra 1, MM muestra 01, MMM muestra Ene y MMMM muestra Enero.
dd: `d` representa la fecha en el mes. `dd` significa que se debe incluir el cero inicial para un mes de un solo dígito
yyyy: `y` representa el año. `yy` genera un año de dos dígitos y `aaaa` genera un año de cuatro dígitos.
hh: `h` representa la hora. Use `hh` para incluir el cero inicial si está generando una hora de un solo dígito.
mm: `m` representa el minuto.

¿Puedes averiguar cuál de estas líneas arrojará una excepción?

```
4: DateTimeFormatter f = DateTimeFormatter.ofPattern("hh:mm");
5: f.format(dateTime);
6: f.format(date);
7: f.format(time);
```

Tenemos h para hora y m parta minuto. Por lo tanto, la línea 6 lanzará una excepción.

6.5. Parsing Dates and Times

Al igual que el método `format()`, el método `parse()` también toma un formateador. Si no especifica uno, usa el predeterminado para ese tipo:

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", f);  // using custom format
LocalTime time = LocalTime.parse("11:22");
System.out.println(date);           // 2015-01-02
System.out.println(time);           // 11:22
```

7. Review Questions

7.1.

What is output by the following code? (Choose all that apply)

```
1: public class Fish {
2:  public static void main(String[] args) {
3:    int numFish = 4;
4:    String fishType = "tuna";
5:    String anotherFish = numFish + 1;
6:    System.out.println(anotherFish + " " + fishType);
7:    System.out.println(numFish + " " + 1);
8:  } }
```

A. `4 1`
B. `41`
C. `5`
D. `5 tuna`
E. `5tuna`
F. `51tuna`
G. The code does not compile.

7.2.

What is the result of the following code?

```
7: StringBuilder sb = new StringBuilder();
8: sb.append("aaa").insert(1, "bb").insert(4, "ccc");
9: System.out.println(sb);
```

A. `abbaaccc`
B. `abbaccca`
C. `bbaaaccc`
D. `bbaaccca`
E. An exception is thrown.
F. The code does not compile.

7.3.

What is the result of the following code? (Choose all that apply)

```
13: String a = "";
14: a += 2;
15: a += 'c';
16: a += false;
17: if ( a == "2cfalse") System.out.println("==");
18: if ( a.equals("2cfalse")) System.out.println("equals");
```

A. Compile error on line 14.
B. Compile error on line 15.
C. Compile error on line 16.
D. Compile error on another line.
E. `==`
F. `equals`
G. An exception is thrown.

7.4.

Which of the following can replace line 4 to print `"avaJ"`? (Choose all that apply)

```
3: StringBuilder puzzle = new StringBuilder("Java");
4: // INSERT CODE HERE
5: System.out.println(puzzle);
```

A. `puzzle.reverse();`
B. `puzzle.append("vaJ$").substring(0, 4);`
C. `puzzle.append("vaJ$").delete(0, 3).deleteCharAt(puzzle.length() - 1);`
D. `puzzle.append("vaJ$").delete(0, 3).deleteCharAt(puzzle.length());`
E. None of the above.

7.5.

What is the result of the following?

```
4: List<Integer> list = Arrays.asList(10, 4, -1, 5);
5: Collections.sort(list);
6: Integer array[] = list.toArray(new Integer[4]);
7: System.out.println(array[0]);
```

A. `-1`
B. `10`

C. Compiler error on line 4.
D. Compiler error on line 5.
E. Compiler error on line 6.
F. An exception is thrown.

7.6.

What is the output of the following code?

```
LocalDate date = LocalDate.parse("2018-04-30",
                  DateTimeFormatter.ISO_LOCAL_DATE);
date.plusDays(2);
date.plusHours(3);
System.out.println(date.getYear() + " " + date.getMonth()
                  + " " + date.getDayOfMonth());
```

A. `2018 APRIL 2`
B. `2018 APRIL 30`
C. `2018 MAY 2`
D. The code does not compile.
E. A runtime exception is thrown.

7.7.

What is the output of the following code?

```
LocalDateTime d = LocalDateTime.of(2015, 5, 10, 11, 22, 33);
Period p = Period.of(1, 2, 3);
d = d.minus(p);
DateTimeFormatter f = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(d.format(f));
```

A. `3/7/14 11:22 AM`
B. `5/10/15 11:22 AM`
C. `3/7/14`
D. `5/10/15`
E. `11:22 AM`
F. The code does not compile.
G. A runtime exception is thrown.