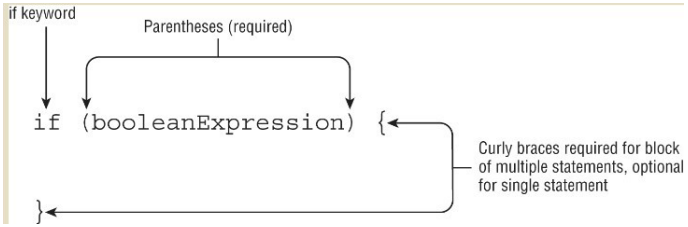
Core APIs

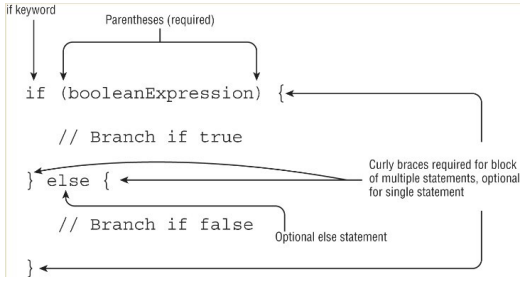
sentencia *IF*

El objetivo de una instrucción de toma de decisiones puede ser una sola instrucción o un bloque de instrucciones.




```
1 // sentencia de una sola linea cuando solo hay una ejecución después del IF
2 if (a > b)
3  a++;
4
5 // sentencia de múltiples lineas, cuando queremos hacer mas de una sentencia
6 if (a > 3){
7  a++;
8 }
```

la sentencia ***IF-ELSE*** es utilizada para evitar tener que hacer 2 IF




```
1 if(hora < 11){
2  System.out.println("adios");
3 } else System.out.println("hola");
```

Reducción de código de coincidencia de patrones: Java 16 introdujo oficialmente la coincidencia de patrones con las instrucciones ***if*** y el operador ***instanceof***.

Para declarar una cadena podemos hacerlo con:
String name = "Fluffy";
String name = new String("Fluffy");
string name = """
  Fluffy""";

Al ser un string una secuencia de caracteres este es una implementación de la interfaz CharSequence.

concatenacion: consiste en la unión de cadenas y tiene las siguientes reglas:
- si ambos operadores son numericos "+" significa adicion
- si ambos operadores son cadenas "+" significa concatenacion
- las expresiones se evaluan de izquierda a derecha

```
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
System.out.println("c" + 1 + 2); //c12
System.out.println("c" + null); //cnull


int a1 = 3;
String a2  = "4";
System.out.println(1 + 2 + a1 + a2); //64

var s = "1"; // s = "1"
s += "2"; // s = "12"
s += 3; // s = "123"
```

Debemos tener en claro que String tiene muchos metodos y que Java lo considera una secuencia de caracteres y a su vez es inmutable, esto significa que no puede ser modificado despues de creado por lo cual el uso de cualquier metodo creara un nuevo objeto String con el resultado.

calculando el tamaño de la adena on "length()"

```
var name = "animals";
System.out.println(name.length()); //7
```

obteniendo un unico caracter con "charAt()"

```
var name = "animals";
System.out.println(name.charAt(0)); //a
```

cuando se pasa un indice que no existe en la cadena este devuelve una excepcion

encontrando un indice con "indexOf()", este busca el primer indice que coincida con el caracter o cadena indicada. ya que un char puede ser pasado como un numero tambien podemos ver este tipo de parametros pero en el examen siempre lo asocian con un parametro denominado "ch". los posibles parametros de este metodo son:

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

ejemplo:

```
var name = "animals";

System.out.println(name.indexOf('a')); // 0
System.out.println(name.indexOf('al')); // 4
System.out.println(name.indexOf('a',4)); // 4
System.out.println(name.indexOf('al',5)); // -1
```

obteniendo una subcadena con el metodo "substring()" en el cual marcamos hasta donde extraeremos la cadena sin contar la posicion final. se puede indicar el parametro inicial o final o solo el final.

```
public int substring(inicio, final)

var name = "animals";
System.out.println(name.substring(3));   // mals
System.out.println(name.substring(name.indexOf('m'))); // mals
System.out.println(name.substring(3, 4));  // m
System.out.println(name.substring(3, 7));  // mals
System.out.println(name.substring(3, 3));  // vacio
System.out.println(name.substring(3, 2));  // excepcion
```

tambien tenemos los metodos "toLowerCase()" que convierte todo el texto a minusculas y "toUpperCase()" que lo hace a mayusculas

```
var name = "animals";
System.out.println(name.toUpperCase());   // ANIMALS
System.out.println("Abc123".toLowerCase());   // abc123
```

El método equals() verifica si dos objetos de tipo String contienen exactamente los mismos caracteres en el mismo orden. Por otro lado, el método equalsIgnoreCase() hace lo mismo, pero ignora si los caracteres están en mayúsculas o minúsculas.

```
public boolean equals(Object obj)
public boolean equalsIgnoreCase(String str)

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

Sobrescribiendo toString(), equals(Object) y hashCode()

Saber cómo sobrescribir correctamente estos métodos es fundamental en Java. Aunque ya no es un requisito obligatorio para las certificaciones, sigue siendo una práctica esencial para cualquier desarrollador Java.

- toString(): Se invoca cuando intentas imprimir un objeto o concatenarlo con una cadena. Generalmente se sobrescribe para proporcionar una descripción única del objeto basada en sus atributos.

- equals(Object): Se utiliza para comparar objetos. La implementación por defecto simplemente compara las referencias, pero es común sobrescribirla para realizar comparaciones más significativas basadas en los atributos del objeto.

- hashCode(): Siempre que sobrescribas equals(Object), debes sobrescribir hashCode() de manera consistente. Esto significa que si a.equals(b) es verdadero, entonces a.hashCode() == b.hashCode() también debe ser verdadero.

A menudo, necesitas buscar dentro de una cadena más larga para determinar si una subcadena está contenida en ella. Los métodos startsWith() y endsWith() verifican si el valor proporcionado coincide con el inicio o el final de la cadena. El método contains() es menos específico; busca coincidencias en cualquier parte de la cadena.

```
public boolean startsWith(String prefix)
public boolean endsWith(String suffix)
public boolean contains(CharSequence charSeq)

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

El método replace() realiza una búsqueda y reemplazo simple en la cadena. Existe una versión que acepta caracteres como parámetros y otra que acepta secuencias de caracteres. Las firmas de los métodos son las siguientes:

public String replace(char oldChar, char newChar)
public String replace(CharSequence target, CharSequence replacement)

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

Primer ejemplo: Aquí, reemplazamos todas las ocurrencias del carácter 'a' por 'A'. utiliza el reemplazo por caracter
Segundo ejemplo: En este caso, reemplazamos todas las ocurrencias de la subcadena "a" por "A". aqui utilizamos el reemplazo por cadena de caracteres

eliminación de espacios en blanco: Estos métodos se utilizan para eliminar los espacios en blanco al principio y/o al final de una cadena. Los espacios en blanco incluyen espacios, tabulaciones (\t) y saltos de línea (\n).

public String strip()
public String stripLeading()
public String stripTrailing()
public String trim()

- strip(): Elimina todos los espacios en blanco al principio y al final de la cadena.
- stripLeading(): Elimina los espacios en blanco al principio de la cadena.
- stripTrailing(): Elimina los espacios en blanco al final de la cadena.
- trim(): Es similar a strip(), pero no admite todos los caracteres Unicode.

obs: por ejemplo un caracter Unicode que representa un espacio es char ch = '\u2000';

System.out.println(" abc ".strip()); // abc
System.out.println("\t abc\n".strip()); // abc

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

obs: \t: Representa un carácter de tabulación.

indent(): Agrega un número específico de espacios al inicio de cada línea. Si se pasa un número negativo, intenta eliminar esa cantidad de espacios.

stripIndent(): Elimina la sangría inicial de cada línea.
Normalización de espacios en blanco:

El método indent() también normaliza los espacios en blanco. Esto significa que: Se agrega un salto de línea al final de la cadena si no existe.
Todos los saltos de línea se convierten al formato estándar \n.

El método stripIndent() es útil cuando una cadena se ha construido concatenando partes más pequeñas en lugar de usar un bloque de texto. Este método elimina todos los espacios en blanco adicionales al principio de cada línea. Esto significa que todas las líneas que contienen texto se alinean al principio, eliminando la misma cantidad de espacios en blanco al inicio de cada una, y asegurando que el primer carácter de cada línea no sea un espacio en blanco. Al igual que indent(), convierte los saltos de línea (\r\n) en el formato estándar (\n). Sin embargo, a diferencia de indent(), stripIndent() no agrega un salto de línea al final si falta."

imagen01 -> reglas de indent() y stripIndent()

```
var block = """
     a
     b
     c""";
var concat = " a\n"
    + "b\n"
    + "c";

System.out.println(block.length()); // 6
System.out.println(concat.length()); // 9
System.out.println(block.indent(1).length()); // 10
System.out.println(concat.indent(-1).length()); // 7
System.out.println(concat.indent(-4).length()); // 6
System.out.println(concat.stripIndent().length()); // 6
```

Traduciendo Escapes:
Cuando escapamos caracteres, usamos una sola barra invertida. Por ejemplo, \t es un tabulador. Si no queremos este comportamiento, agregamos otra barra invertida para escapar la barra invertida, por lo que \t es la cadena literal \t. El método translateEscapes() toma estos literales y los convierte en el carácter escapado equivalente. La firma del método es la siguiente:

public String translateEscapes()

```
var str = "1\\t2";
System.out.println(str);           // 1\t2
System.out.println(str.translateEscapes()); // 1  2
```

este metodo permite esapar \t (tab), \n (nueva linea), \s (esapcio) \" (comillas dobles) y \' (comilla simple)

revisión de cadenas blancas y vacias:
Java proporciona métodos de conveniencia para comprobar si una String tiene una longitud de cero o contiene solo caracteres de espacio en blanco. Las firmas de los métodos son las siguientes:

```
public boolean isEmpty()
public boolean isBlank()

System.out.println(" ".isEmpty()); // false - tiene un espacio en blanco
System.out.println("".isEmpty()); // true - no hay ningun caracter
System.out.println(" ".isBlank()); // true - no hay caracteres ademas del espacio en blanco.
System.out.println("".isBlank()); // true - no hay caracteres ademas del espacio en blanco.
```

Formateando Valores:
Hay métodos para formatear valores String usando banderas de formateo. Dos de los métodos toman la cadena de formato como parámetro, y el otro usa una instancia para ese valor.

Los parámetros del método se utilizan para construir una String formateada en una sola llamada al método, en lugar de hacerlo mediante muchas operaciones de formato y concatenación. Devuelven una referencia a la instancia sobre la que se llaman para que las operaciones se puedan encadenar juntas. Las firmas de los métodos son las siguientes:

```
public static String format(String format, Object args...)
public static String format(Locale loc, String format, Object args...)
public String formatted(Object args...)

var name = "Kate";
var orderId = 5;

// Todos imprimen: Hello Kate, order 5 is ready

System.out.println("Hello "+name+", order "+orderId+" is ready");
System.out.println(String.format("Hello %s, order %d is ready",
   name, orderId));
System.out.println("Hello %s, order %d is ready"
   .formatted(name, orderId));
```

En las operaciones format() y formatted(), los parámetros se insertan y formatean mediante símbolos en el orden en que se proporcionan en el Vargs.

Formateo comunes:

%s Se aplica a cualquier tipo, comúnmente valores String
%d Se aplica a valores enteros como int y long
%f Se aplica a valores de punto flotante como float y double
%n Inserta un salto de línea usando el separador de línea dependiente del sistema

```
var name = "James";
var score = 90.25;
var total = 100;

System.out.println("%s:%n Score: %f out of %d".formatted(name, score, total));
```

la salida es:

```
James:
Score: 90.250000 out of 100
```

La mezcla de tipos de datos puede causar excepciones en tiempo de ejecución. Por ejemplo, lo siguiente lanza una excepción porque se usa un número de punto flotante cuando se espera un valor entero:

```
var str = "Food: %d tons".formatted(2.0);
IllegalFormatConversionException
```

las banderas ya definidas se pueden ajustar mas; En el ejemplo anterior, el número de punto flotante se imprimió como 90.250000. Por defecto, %f muestra exactamente seis dígitos después del decimal. Si deseas mostrar solo un dígito después del decimal, puedes usar %.1f en lugar de %f. El método format() se basa en el redondeo en lugar de truncar al acortar números.

Por ejemplo, 90.250000 se mostrará como 90.3 (no 90.2) cuando se pase a format() con %.1f. El método format() también admite dos características adicionales. Puedes especificar la longitud total de la salida usando un número antes del símbolo decimal. Por defecto, el método rellenará el espacio vacío con espacios en blanco. También puedes rellenar el espacio vacío con ceros colocando un solo cero antes del símbolo decimal. Los siguientes ejemplos usan corchetes, [], para mostrar el inicio/fin del valor formateado:

```
var pi = 3.141592653589;
System.out.format("[%6f]",pi);    // [3.141593]
System.out.format("[%12.8f]",pi);  // [ 3.14159265]
System.out.format("[%012f]",pi);   // [00003.141593]
System.out.format("[%12.2f]",pi);  // [       3.14]
System.out.format("[%.3f]",pi);    // [3.142]
```

Encadenamiento de Métodos:

```
var start = "AniMaL ";
var trimmed = start.trim();        // "AniMaL"
var lowercase = trimmed.toLowerCase(); // "animal"
```

```
var result = lowercase.replace('a', 'A'); // "AnimAl"
System.out.println(result);
```

Esto es solo una serie de métodos String. Cada vez que se llama a uno, el valor devuelto se coloca en una nueva variable. Hay cuatro valores String a lo largo del camino, y se muestra AnimAl como salida.
Sin embargo, en el examen, hay una tendencia a comprimir tanto código como sea posible en un espacio pequeño. Verás código usando una técnica llamada "encadenamiento de métodos". Aquí hay un ejemplo:

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

Este código es equivalente al ejemplo anterior. También crea cuatro objetos String y muestra AnimAl. Para leer código que usa encadenamiento de métodos, empieza por la izquierda y evaluar el primer método. Luego llamar al siguiente método sobre el valor devuelto por el primer método. Continúa así hasta que llegues al punto y coma.

```
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);
```

En la línea 5, establecemos que a apunte a "abc" y nunca apuntó a nada más. Como ninguno del código en las líneas 6 y 7 cambia a, el valor permanece "abc".
Sin embargo, b es un poco más complicado. La línea 6 hace que b apunte a "ABC", lo cual es sencillo. En la línea 7, tenemos encadenamiento de métodos. Primero se llama a "ABC".replace("B", "2"). Esto devuelve "A2C". Luego, se llama a "A2C".replace('C', '3'), lo que devuelve "A23".

Usando la Clase StringBuilder:

```
10: String alpha = "";
11: for(char current = 'a'; current <= 'z'; current++)
12:   alpha += current;
13: System.out.println(alpha);
```

La String vacía en la línea 10 se instancia, y luego la línea 12 añade una "a". Sin embargo, como el objeto String es inmutable, se asigna un nuevo objeto String a alpha, y el objeto "" se vuelve elegible para la recolección de basura.
La siguiente vez que se pasa por el bucle, alpha se asigna a un nuevo objeto String "ab", y el objeto "a" se vuelve elegible para la recolección de basura.
La siguiente iteración asigna alpha a "abc", y el objeto "ab" se vuelve elegible para la recolección de basura, y así sucesivamente.

Esta secuencia de eventos continúa, y después de 26 iteraciones a través del bucle, un total de 27 objetos son instanciados, la mayoría de los cuales son inmediatamente elegibles para la recolección de basura.

Esto es muy ineficiente. Afortunadamente, Java tiene una solución. La clase StringBuilder crea una String sin almacenar todos esos valores String intermedios. A diferencia de la clase String, StringBuilder no es inmutable.

```
15: StringBuilder alpha = new StringBuilder();
16: for(char current = 'a'; current <= 'z'; current++)
17:  alpha.append(current);
18: System.out.println(alpha);
```

En la línea 15, se instancia un nuevo objeto StringBuilder. La llamada a append() en la línea 17 añade un carácter al objeto StringBuilder cada vez que pasa por el bucle for, añadiendo el valor de current al final de alpha. Este código reutiliza el mismo StringBuilder sin crear una String intermedia cada vez.

Mutabilidad y Encadenamiento:

Seguramente notaste esto del ejemplo anterior, pero StringBuilder no es inmutable. De hecho, le dimos 27 valores diferentes en el ejemplo (un espacio en blanco más cada letra del alfabeto). El examen probablemente intentará engañarte con respecto a la mutabilidad de String y StringBuilder.

El encadenamiento hace esto aún más interesante. Cuando encadenamos llamadas a métodos String, el resultado era una nueva String con la respuesta. El encadenamiento de métodos StringBuilder no funciona así. En su lugar, el StringBuilder cambia su propio estado y devuelve una referencia a sí mismo. Veamos un ejemplo para aclarar esto:

```
4: StringBuilder sb = new StringBuilder("start");
5: sb.append("+middle"); // sb = "start+middle"
6: StringBuilder same = sb.append("+end"); // "start+middle+end"
```

La línea 5 añade texto al final de sb. También devuelve una referencia a sb, que se ignora. La línea 6 también añade texto al final de sb y devuelve una referencia a sb. Esta vez la referencia se almacena en same. Esto significa que sb y same apuntan al mismo objeto e imprimirían el mismo valor.

El examen no siempre hará que el código sea fácil de leer teniendo solo un método por línea. ¿Qué crees que imprime este 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);
```

¿Dijiste que ambos imprimen "abcdefg"? Bien. Solo hay un objeto StringBuilder aquí. Sabemos eso porque new StringBuilder() se llama solo una vez. En la línea 5, en la línea 6, ambas variables siguen apuntando al mismo objeto, que ahora tiene un valor de "abcdefg". Asignar el valor de nuevo a 'b' no tiene ningún efecto, ya que 'b' ya está apuntando a ese mismo StringBuilder.

Creación de un StringBuilder

Hay tres formas de construir un StringBuilder:

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

El primer ejemplo crea un StringBuilder que contiene una secuencia vacía de caracteres y asigna a sb1 para que lo apunte. El segundo crea un StringBuilder que contiene un valor específico y asigna a sb2 para que lo apunte. Los dos primeros ejemplos permiten que Java gestione los detalles de la implementación. El último ejemplo le indica a Java que tenemos alguna idea de cuál será el valor final y que nos gustaría que el StringBuilder reserve una cierta capacidad, o número de espacios, para caracteres.

Metodos comunes de un StringBuilder:

```
var 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);
```

La respuesta correcta es anim 7 s. El método indexOf() devuelve 0 y 4, respectivamente. El método substring() devuelve la cadena que comienza con el índice 0 y termina justo antes del índice 4.

El método length() devuelve 7 porque ese es el número de caracteres en el StringBuilder, en lugar de un índice. Finalmente, charAt() devuelve el carácter en el índice 6. Aquí comenzamos con 0 porque nos referimos a índices.

Nota que substring() devuelve un String en lugar de un StringBuilder. Por eso sb no se modifica. El método substring() es simplemente un método que consulta el estado del StringBuilder.

Agregar valores (Appending Values):

El método append() es, con mucho, el más utilizado en StringBuilder. De hecho, se usa tan frecuentemente que simplemente comenzamos a usarlo sin comentar al respecto. Afortunadamente, este método hace exactamente lo que parece: agrega el parámetro al StringBuilder y devuelve una referencia al StringBuilder actual. Una de las firmas del método es la siguiente:

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

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

¿Te gusta esta forma de encadenar métodos? El método append() se llama directamente después del constructor. Al contar con todas estas firmas de métodos, puedes simplemente llamar a append() sin necesidad de convertir tu parámetro a un String primero.

Insertar datos (Inserting Data):

El método insert() agrega caracteres al StringBuilder en el índice solicitado y devuelve una referencia al StringBuilder actual. Al igual que append(), existen muchas firmas del método para diferentes tipos. Aquí tienes una:

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

Presta atención al desplazamiento (offset) en estos ejemplos. Es el índice donde queremos insertar el parámetro solicitado.

```
var sb = new StringBuilder("animals");
sb.insert(7, "-");   // sb = animals-
sb.insert(0, "-");   // sb = -animals-
sb.insert(4, "-");   // sb = -ani-mals-
System.out.println(sb);
```

La línea 4 indica insertar un guion (-) en el índice 7, que resulta ser el final de la secuencia de caracteres. La línea 5 indica insertar un guion en el índice 0, que es el principio de la secuencia. Finalmente, la línea 6 inserta un guion justo antes del índice 4.

Eliminando contenido:

El método delete() es lo opuesto al método insert(). Elimina caracteres de la secuencia y devuelve una referencia al StringBuilder actual. Si solo quieres eliminar un carácter, el método deleteCharAt() es más conveniente. Las firmas de estos métodos son las siguientes:

public StringBuilder delete(int startIndex, int endIndex)
public StringBuilder deleteCharAt(int index)

El siguiente código muestra cómo usar estos métodos:

```
var sb = new StringBuilder("abcdef");
sb.delete(1, 3); // sb = adef
sb.deleteCharAt(5); // excepción
```

Primero, eliminamos los caracteres comenzando desde el índice 1 y terminando justo antes del índice 3. Esto nos da "adef". Luego, le pedimos a Java que elimine el carácter en la posición 5. Sin embargo, el valor restante solo tiene cuatro caracteres de longitud, por lo que lanza una excepción StringIndexOutOfBoundsException.

Reemplazando Porciones:

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

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

```
var builder = new StringBuilder("pigeon dirty");
builder.replace(3, 6, "sty");
System.out.println(builder); // pigsty dirty
```

Primero, Java elimina los caracteres comenzando desde el índice 3 y terminando justo antes del índice 6. Esto nos da "pig dirty". Luego, Java inserta el valor "sty" en esa posición.

En este ejemplo, el número de caracteres que eliminamos y los que insertamos son iguales. Sin embargo, no hay ninguna razón obligatoria para que esto sea así. ¿Qué crees que hace este código?

```
var builder = new StringBuilder("pigeon dirty");
builder.replace(3, 100, "");
System.out.println(builder);
```

Imprime "pig". Recuerda, el método realiza primero una eliminación lógica. El método replace() permite especificar un segundo parámetro que excede el final del StringBuilder.

Invertir (Reversing):

Después de todo esto, es hora de un método fácil. El método reverse() hace exactamente lo que parece: invierte los caracteres en la secuencia y devuelve una referencia al StringBuilder actual. La firma del método es la siguiente:

public StringBuilder reverse()

```
var sb = new StringBuilder("ABC");
sb.reverse();
System.out.println(sb);
```

El resultado será "CBA". Este método no es muy interesante. Tal vez los creadores del examen lo incluyan para animarte a escribir el valor en lugar de confiar en la memoria para los índices.

Trabajando con toString()

La clase Object contiene un método toString() que muchas clases proporcionan como implementación personalizada. La clase StringBuilder es una de estas.

El siguiente código muestra cómo usar este método:

```
var sb = new StringBuilder("ABC");
String s = sb.toString();
```

A menudo, StringBuilder se usa internamente por razones de rendimiento, pero el resultado final necesita ser una cadena (String). Por ejemplo, tal vez se necesite pasar el valor a otro método que espera un String.

Entendiendo la igualdad (Understanding Equality):

En el Capítulo 2, aprendiste a usar == para comparar números y que las referencias de objetos apuntan al mismo objeto. En esta sección, analizamos lo que significa para StringBuilder.

En Java, para determinar si dos objetos son iguales, no solo basta con comparar sus referencias en memoria (usando el operador ==). A veces, necesitamos comparar el contenido de esos objetos. Esto es especialmente relevante cuando hablamos de clases como String y StringBuilder.

Java tiene una característica llamada "String Pool". Cuando creas un objeto String con una cadena literal (entre comillas dobles), Java busca en el String Pool para ver si ya existe un objeto String con ese mismo valor. Si lo encuentra, te devuelve una referencia a ese objeto existente en lugar de crear uno nuevo. Esto ayuda a ahorrar memoria y mejorar el rendimiento.

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

Comparando equals() y ==

==: Compara las referencias de los objetos. Si dos referencias apuntan al mismo objeto en memoria, == devuelve true.

equals(): Compara el contenido de los objetos. La implementación de equals() varía según la clase. Para las clases envolventes (Integer, Double, etc.) y String, equals() compara el valor.

```
var one = new StringBuilder();
var two = new StringBuilder();
var three = one.append("a");

System.out.println(one == two); // false - one y two son objetos diferentes
System.out.println(one == three); // true - three es una referencia al mismo objeto que one después de llamar a append()
```

StringBuilder y equals()

StringBuilder no sobreescribe el método equals() para comparar el contenido. Por defecto, equals() en StringBuilder compara las referencias.
Si quieres comparar el contenido de dos StringBuilder, debes convertirlos a String usando toString() y luego comparar los String.

```
var name = "a";
var builder = new StringBuilder("a");
System.out.println(name == builder); // DOES NOT COMPILE
```

Este código no compila porque estás intentando comparar un String con un StringBuilder. Java no puede comparar directamente objetos de tipos diferentes.

El Pool de Cadenas (String Pool)

Dado que las cadenas (strings) están presentes en todas partes en Java, consumen una gran cantidad de memoria. En algunas aplicaciones de producción, pueden ocupar una gran porción de la memoria total del programa. Java reconoce que muchas cadenas se repiten en el programa y resuelve este problema reutilizando las comunes. El pool de cadenas (también conocido como el intern pool) es una ubicación en la Máquina Virtual de Java (JVM) que almacena todas estas cadenas.

El pool de cadenas contiene valores literales y constantes que aparecen en tu programa. Por ejemplo, "nombre" es un literal y, por lo tanto, entra en el pool de cadenas. El método myObject.toString() devuelve una cadena pero no un literal, así que no entra en el pool.

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

Dado que las cadenas son inmutables y los literales se almacenan en el pool, la JVM crea solo una ubicación en memoria para el literal. Por lo tanto, las variables x e y apuntan al mismo lugar en memoria; por lo tanto, la instrucción imprime true.

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

La primera línea utiliza el pool de cadenas normalmente. La segunda línea dice: "No, JVM, realmente no quiero que uses el pool de cadenas. Por favor, crea un nuevo objeto para mí, aunque sea menos eficiente".

Utilizando el método intern():

También puedes decirle a Java que use el pool de cadenas. El método intern() utilizará un objeto en el pool de cadenas si ya existe uno presente.

public String intern()

```
var name = "Hello World";
var name2 = new String("Hello World").intern();
System.out.println(name == name2); // true
```

Primero, le decimos a Java que use el pool de cadenas normalmente para name. Luego, para name2, le decimos que cree un nuevo objeto usando el constructor pero que lo interné y use el pool de cadenas de todos modos. Dado que ambas variables apuntan a la misma referencia en el pool de cadenas, podemos usar el operador ==.

```
15| var first = "rat" + 1;
16| var second = "r" + "a" + "t" + "1";
17| var 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());
```

Líneas 15 y 16: Ambas son constantes en tiempo de compilación y se colocan automáticamente en el pool de cadenas.

Línea 17: Aquí tenemos un constructor de String. Esto significa que ya no tenemos una constante en tiempo de compilación y third no hace referencia a una llamada en el pool de cadenas.

Líneas 18 y 19: La línea 18 es false porque third no está en el pool. La línea 19 es true porque intern() busca en el pool de cadenas y encuentra que first apunta a la misma cadena.

Línea 20 imprime falso

Arreglos:

Hasta ahora, nos hemos referido a las clases String y StringBuilder como una "secuencia de caracteres". Esto es cierto. Están implementadas usando un arreglo de caracteres. Un arreglo es un área de memoria en el heap con espacio para un número designado de elementos. Un String se implementa como un arreglo con algunas métodos que te pueden ser útiles cuando estás tratando con caracteres específicamente. Un StringBuilder es implementado usando un arreglo donde el objeto del arreglo es reemplazado con uno nuevo, más grande cuando se queda sin espacio para almacenar los caracteres. A diferencia de un String, un arreglo puede ser de cualquier otro tipo de Java. Si no quieres usar un String por alguna razón, podrías usar un arreglo de char primitivos directamente:

```
char[] letters;
```

Esto no sería muy conveniente porque perderíamos todas las propiedades especiales que String nos da, como escribir "Java". Mantén en mente que letters es una variable de referencia y no un primitivo. El tipo char es un primitivo. Pero char es lo que va dentro del arreglo y no el tipo del arreglo en sí. El arreglo en sí es de tipo char[].

Creando un Arreglo de Primitivos

```
int[] numbers = new int[3];
```

Cuando usas este formulario para instanciar un arreglo, todos los elementos se establecen en el valor por defecto para ese tipo. Como aprendiste en el Capítulo 1, el valor por defecto de un int es 0. En la Figura 4.4, puedes ver que el valor por defecto para todos los elementos es 0. También, los índices comienzan con 0 y cuentan hacia arriba, así como lo hacían para un String.

Cuando usas este formulario para instanciar un arreglo, todos los elementos se establecen en el valor por defecto para ese tipo. Como aprendiste en el Capítulo 1, el valor por defecto de un int es 0 (imagen02)

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

En este ejemplo, también creamos un arreglo de enteros de tamaño 3. Esta vez, especificamos los valores iniciales de esos tres elementos en lugar de usar los valores predeterminados.

Java reconoce que esta expresión es redundante. Dado que estás especificando el tipo del arreglo en el lado izquierdo del signo igual, Java ya sabe el tipo. Y dado que estás especificando los valores iniciales, ya sabe el tamaño. Como un atajo, Java te permite escribir esto:

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

Este enfoque se llama un arreglo anónimo. Es anónimo porque no especificas el tipo y el tamaño.

Finalmente, puedes escribir los corchetes ([]) antes o después del nombre, y agregar un espacio es opcional. por ejemplo, lo siguiente es valido:

```
int[] numA1;
int [] numA1;
int []numA1;
int numA1[];
int numA1 [];
```

¿Qué tipos de variables de referencia crees que crea el siguiente código?

```
int[] ids, types;
```

La respuesta correcta es que crea dos variables de tipo int[]

```
int ids[], types;
```

La primera se llama ids[]. Esta es un arreglo de int, llamada ids. La segunda es solo llamada types. No tiene corchetes, así que es un entero regular

Creando un Arreglo con Variables de Referencia

Puedes elegir cualquier tipo de Java para ser el tipo del arreglo. Esto incluye clases que tú mismo crees. Vamos a ver un ejemplo con el tipo de dato predefinido String:

```
String[] bugs = {"cricket", "beetle", "ladybug"};
String[] alias = bugs;

System.out.println(bugs.equals(alias)); // true
System.out.println(bugs.toString()); // [Ljava.lang.String;@160bc7c0

public class Names {
   String[] names = new String[2];
}
```

Es un arreglo porque tiene corchetes.

Es un arreglo de tipo String ya que esa es el tipo mencionado en la declaración.

Tiene dos elementos porque la longitud es 2, pero esos dos espacios actualmente son null pero tienen el potencial de apuntar a un objeto String.

```
3| String[] strings = {"stringValue"};
4| Object[] objects = strings;
5| String[] againStrings = (String[]) objects;
6| againStrings[0] = new StringBuilder(); // ¡NO COMPILA!
7| objects[0] = new StringBuilder(); // ¡Cuidado!
```

- La línea 3 crea un arreglo de tipo String.
- La línea 4 no requiere un cast porque Object es un tipo más amplio que String.
- En la línea 5, se necesita un cast porque estamos moviéndonos a un tipo específico.
- La línea 6 no compila porque String[] solo permite objetos String y StringBuilder no es un String.
- La línea 7 es donde se pone interesante. Desde el punto de vista del compilador, esto está bien. Un objeto StringBuilder claramente puede ir en un Object[]. El problema es que no tenemos actualmente un Object[]. Tenemos un String[] referido desde una variable Object. Al runtime, el código arrojará una ArrayStoreException.

Usando un Arreglo

```
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
```

La línea 4 declara e inicializa el arreglo.
La línea 5 nos dice cuántos elementos puede contener el arreglo.
El resto del código imprime el arreglo.
Nota que los elementos están indexados comenzando con 0. Esto debería ser familiar de String y StringBuilder, que también comienzan a contar con 0.

length como el número de elementos: Después de length no hay paréntesis porque no es un método, es un atributo.

```
String[] mammals = {"monkey", "chimp", "donkey"};
System.out.println(mammals.length); // Imprime 3

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

La respuesta es 6. Aunque los seis elementos del arreglo son null, todavía hay seis de ellos. El atributo length no considera lo que hay en el arreglo; solo considera cuántos slots han sido asignados.

Es muy común usar un loop cuando lees o escribes en un arreglo. Este loop asigna a cada elemento de numbers un valor cinco más grande que el índice actual:

```
var numbers = new int[10];
for (int i = 0; i < numbers.length; i++) {
   numbers[i] = i + 5;
}
```

El primero está intentando ver si sabes que los índices empiezan con 0. Ya que tenemos 10 elementos en nuestro arreglo, esto significa que solo numbers[0] hasta numbers[9] son válidos.

Ordenando Arreglos:

Al igual que StringBuilder te permitía pasar casi cualquier cosa a append(), puedes pasar casi cualquier arreglo a Arrays.sort().

Para usar Arrays, necesitas importar. Tienes dos opciones:

```
import java.util.*; // Importa todo el paquete, incluyendo Arrays
import java.util.Arrays; // Importa solo Arrays
```

Hay una excepción, aunque no suele aparecer mucho en los exámenes. Puedes escribir java.util.Arrays cada vez que lo uses en la clase en lugar de especificarlo como una importación.

```
int[] numbers = {6, 9, 1};
Arrays.sort(numbers);
for (int i = 0; i < numbers.length; i++) {
   System.out.print(numbers[i] + " ");
}
```

El resultado es 1 6 9, como esperarías. Nota que si hubiéramos iterado directamente sobre el arreglo para imprimir los valores, obtendríamos el molesto hash de [I@2bd9c3e7. Alternativamente, podríamos haber usado Arrays.toString(numbers) en lugar del loop. Eso habría impreso [1, 6, 9].

```
String[] strings = {"10", "9", "100"};
Arrays.sort(strings);
for (String s : strings) {
   System.out.print(s + " ");
}
```

El resultado podría no ser el que esperas. Esto imprime 10 100 9. El problema es que los String se ordenan alfabéticamente, y los números van antes de las letras, y las mayúsculas antes de las minúsculas. En el Capítulo 9, "Colecciones y Genéricos", aprenderás a crear órdenes de clasificación personalizados usando algo llamado un comparador.

Búsqueda en Arreglos Ordenados: Búsqueda Binaria

Java también proporciona una forma conveniente de buscar elementos en un arreglo, pero solo si el arreglo ya está ordenado. La Tabla 4.3 resume las reglas para la búsqueda binaria.

Escenario	Resultado
Elemento objetivo encontrado en arreglo ordenado	Índice de la coincidencia
Elemento objetivo no encontrado en arreglo ordenado	Valor negativo que muestra uno menos que el negativo del índice donde debería insertarse un elemento para preservar el orden ordenado
Arreglo no ordenado	Resultado indefinido

```
int[] numbers = {2, 4, 6, 8};
System.out.println(Arrays.binarySearch(numbers, 2)); // 0
System.out.println(Arrays.binarySearch(numbers, 4)); // 1
System.out.println(Arrays.binarySearch(numbers, 1)); // -1
```

Línea 3: Crea un arreglo ordenado de números.
Líneas 4-6: Usan Arrays.binarySearch para buscar diferentes números en el arreglo.
Línea 4: Encuentra el número 2 en el índice 0.
Línea 5: Encuentra el número 4 en el índice 1.
Línea 6: El número 1 no está en el arreglo, por lo que devuelve -1, indicando que debería insertarse en el índice 0 para mantener el orden.

Usando compare()

Primero, necesitas saber qué significa el valor de retorno. No necesitas conocer los valores exactos de retorno, pero debes saber lo siguiente:

Un número negativo significa que el primer arreglo es más pequeño que el segundo.
Un cero significa que los arreglos son iguales.
Un número positivo significa que el primer arreglo es más grande que el segundo.

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

Este código imprime un número negativo. Debería ser bastante intuitivo que 1 es más pequeño que 2, haciendo que el primer arreglo sea más pequeño.

Ahora que sabes cómo comparar un solo valor, veamos cómo comparar arreglos de diferentes longitudes:

- Si ambos arreglos tienen la misma longitud y los mismos valores en cada posición, se devuelve cero.

- Si todos los elementos son los mismos pero el segundo arreglo tiene elementos adicionales al final, se devuelve un número negativo.

- Si todos los elementos son los mismos, pero el primer arreglo tiene elementos adicionales al final, se devuelve un número positivo.

- Si el primer elemento que difiere es más pequeño en el primer arreglo, se devuelve un número negativo.

- Si el primer elemento que difiere es más grande en el primer arreglo, se devuelve un número positivo.

Estas son algunas reglas adicionales que se aplican no solo al método compare() que hemos visto, sino también al método compareTo(), que se introduce en el Capítulo 8 sobre Lambdas e Interfaces Funcionales:

- null es menor que cualquier otro valor: Si comparas cualquier objeto con null, null siempre se considerará menor.

- Para números, se aplica el orden numérico normal: Los números se comparan de la manera que esperarías: 1 es menor que 2, -5 es menor que 3, etc.
Para cadenas, una es menor si es prefijo de otra: Por ejemplo, "casa" es menor que "castillo" porque "casa" es un prefijo de "castillo".

Para cadenas o caracteres, los números son menores que las letras: Si comparas un número con una letra, el número siempre será considerado menor. Por ejemplo, "1" es menor que "a".

Para cadenas o caracteres, las mayúsculas son menores que las minúsculas: En el orden alfabético, las letras mayúsculas preceden a las minúsculas. Por ejemplo, "A" es menor que "a".

imagen03 -> ejemos de arrays.compare()

Usando mismatch()

Ahora que estás familiarizado con compare(), es momento de aprender sobre mismatch(). Si los arrays son iguales, mismatch() devuelve -1. De lo contrario, devuelve el primer índice donde difieren. ¿Puedes deducir qué imprimirán estos casos?

```
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}));
```

En el primer ejemplo, los arrays son iguales, por lo que el resultado es -1. En el segundo ejemplo, los elementos en la posición 0 no son iguales, así que el resultado es 0. En el tercer ejemplo, los elementos en la posición 0 son iguales, así que seguimos comparando. El elemento en el índice 1 no es igual porque un array tiene un elemento en el índice 1 y el otro no. Por lo tanto, el resultado es 1.

imagen04 -> comparación entre "equals", "compare" y "mismatch"

Usando Métodos con Varargs:

Cuando creas un array, se ve como lo que hemos visto hasta ahora, hay otra forma en que puede verse. 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
```

El tercer ejemplo usa una sintaxis llamada varargs (argumentos variables). Por ahora, solo necesitas saber que puedes usar una variable definida usando varargs como si fuera un array normal. Por ejemplo, args.length y args[0] son válidos.

Trabajando con Arrays Multidimensionales:

Los arrays son objetos, y los componentes del array pueden ser objetos.

Los separadores múltiples de array son todo lo que se necesita para declarar arrays con múltiples dimensiones. Puedes ubicarlos con el tipo o nombre de variable en la declaración:

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

Los primeros dos ejemplos no son sorprendentes y declaran un array bidimensional (2D). El tercer ejemplo también declara un array 2D. No hay buena razón para usar este estilo excepto para confundir a los lectores. El ejemplo final declara dos arrays en la misma línea. Sumando los corchetes, vemos que vars4 es un array 2D y space es un array 3D. De nuevo, no hay razón para usar este estilo excepto para confundir.

Puedes especificar el tamaño de tu array multidimensional en la declaración:

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

El resultado es un array "rectangle" con tres elementos, cada uno referenciando un array de dos elementos.

rectangle[0][1] = "set"; -> si asingamos esto tendriamos otros valores null dentro del array de 2 dimensiones.

```
int[][] diffSize = {{1,4},{3},{9,8,7}}; // tambien podemos declarar arrays asimetricos (irregulaes)
```

otra forma de declarar este tipo de arrays asimetricos es:

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

La operación más común en un array multidimensional es recorrerlo. Este ejemplo imprime un array 2D:

```
var 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] + " "); // imprime elemento
  System.out.println();                 // tiempo para nueva fila
}
```

Aunque podemos reducirlo utilizando el for mejorador

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

Encontrando el Mínimo y Máximo:

Los métodos min() y max() comparan dos valores y devuelven uno de ellos. Las firmas de método para min() son:

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

Hay cuatro métodos sobrecargados, así que siempre tienes una API disponible con el mismo tipo. Cada método devuelve el menor entre a o b. El método max() funciona igual, excepto que devuelve el valor mayor.

El siguiente ejemplo muestra cómo usar estos métodos:

```
int first = Math.max(3, 7); // 7
int second = Math.min(7, -9); // -9
```

La primera línea devuelve 7 porque es mayor. La segunda línea devuelve -9 porque es menor.

Redondeando Números:

El método round() elimina la parte decimal del valor, eligiendo el siguiente número mayor cuando corresponde. Si la parte fraccionaria es .5 o mayor, redondeamos hacia arriba.

Las firmas del método round() son:

```
public static long round(double num)
public static int round(float num)
```

Hay dos métodos sobrecargados para asegurar que haya espacio suficiente para almacenar un double redondeado si es necesario. Ejemplos de uso:

```
long low = Math.round(123.45);    // 123
long high = Math.round(123.50);   // 124
int fromFloat = Math.round(123.45f); // 123
```

La primera línea devuelve 123 porque .45 es menor que la mitad. La segunda línea devuelve 124 porque la parte fraccionaria es justo la mitad. La última línea muestra que un float explícito activa la firma del método que devuelve un int.

Determinando el Techo y el Piso:

El método ceil() toma un valor double. Si es un número entero, devuelve el mismo valor. Si tiene algún valor fraccionario, redondea hacia arriba al siguiente número entero. En contraste, el método floor() descarta cualquier valor después del decimal.

Las firmas de los métodos son:

```
public static double ceil(double num)
public static double floor(double num)
```

Ejemplo de uso:

```
double c = Math.ceil(3.14);  // 4.0
double f = Math.floor(3.14); // 3.0
```

La primera línea devuelve 4.0 porque cuatro es el entero inmediatamente mayor. La segunda línea devuelve 3.0 porque es el entero inmediatamente menor.

Calculando Exponentes:

El método pow() maneja exponentes. Como recordarás de matemáticas básicas, 3^2 significa tres al cuadrado, que es 3 * 3 o 9. También se permiten exponentes fraccionarios. Dieciséis a la potencia 0.5 significa la raíz cuadrada de 16, que es 4. (No te preocupes, no tendrás que calcular raíces cuadradas en el examen).

La firma del método es:

```
public static double pow(double number, double exponent)
```

Ejemplo de uso:

```
double squared = Math.pow(5, 2); // 25.0
```

Nota que el resultado es 25.0 en lugar de 25 ya que es un double. No te preocupes; el examen no te pedirá hacer matemáticas complicadas.

Generando Números Aleatorios:

El método random() devuelve un valor mayor o igual a 0 y menor que 1. La firma del método es:

```
public static double random()
```

```
double num = Math.random();
```

Como es un número aleatorio, no podemos saber el resultado por adelantado. Sin embargo, podemos descartar ciertos números. Por ejemplo, no puede ser negativo porque eso es menor que 0. No puede ser 1.0 porque eso no es menor que 1.

NOTA: Aunque no está en el examen, es común usar la clase Random para generar números pseudo-aleatorios. Permite generar números de diferentes tipos.

Trabajando con Fechas y Tiempos

Java proporciona varias APIs para trabajar con fechas y tiempos. Existe una clase antigua java.util.Date, pero no está en el examen. Necesitas importar para usar las clases modernas de fecha y tiempo:

import java.time.*; // importar clases de tiempo

Día vs. Fecha
En inglés americano, "date" representa dos conceptos: la combinación mes/día/año cuando algo ocurrió (como 1 de enero de 2000) o el día del mes (como "Hoy es día 6"). Las palabras "day" y "date" se usan como sinónimos. Ten cuidado con esto en el examen.

Creando Fechas y Tiempos

En el mundo real, hablamos de fechas y zonas horarias asumiendo que la otra persona está cerca. Por ejemplo, "Te llamaré a las 11:00 el martes" significa lo mismo para ambos si estamos en la misma zona. Pero si estoy en Nueva York y tú en California, debemos ser más específicos debido a la diferencia de 3 horas. Dirías "Te llamaré a las 11:00 EST el martes".

Al trabajar con fechas y tiempos, decide primero cuánta información necesitas. El examen da cuatro opciones:

- LocalDate: Solo fecha, sin hora ni zona horaria. Ejemplo: tu cumpleaños este año.
- LocalTime: Solo hora, sin fecha ni zona horaria. Ejemplo: medianoche.
- LocalDateTime: Fecha y hora, sin zona horaria. Ejemplo: "medianoche de Año Nuevo".
- ZonedDateTime: Fecha, hora y zona horaria. Ejemplo: "conferencia a las 9:00 a.m. EST".

Se obtienen instancias usando un método estático:
```java
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
System.out.println(ZonedDateTime.now());
```

Cada clase tiene un método now() que da la fecha/hora actual. La salida dependerá del momento y lugar. En Estados Unidos, el 25 de octubre a las 9:13 a.m. mostraría:

2021-10-25
09:13:07.768
2021-10-25T09:13:07.768
2021-10-25T09:13:07.769-05:00[America/New_York]

La salida varía según el tipo de información. Muestra solo fecha, solo hora, ambas (separadas por T), o todo incluyendo zona horaria y offset GMT.

creando fechas en java

Veamos cómo crear fechas y horas en Java 17. Para empezar, crearemos solo una fecha sin hora. Ambas formas de crear la fecha son válidas:

public static LocalDate of(int year, int month, int dayOfMonth)
public static LocalDate of(int year, Month month, int dayOfMonth)

En ambos casos, se proporciona el año, el mes y el día. Aunque es recomendable utilizar las constantes de Month (para hacer el código más legible), puedes pasar el número del mes directamente.

var date1 = LocalDate.of(2022, Month.JANUARY, 20);
var date2 = LocalDate.of(2022, 1, 20);

Al crear un objeto de tiempo en Java, tienes la flexibilidad de especificar el nivel de detalle que desees. Puedes incluir solo la hora y los minutos, o agregar segundos e incluso nanosegundos para una precisión extrema (un nanosegundo es una milmillonésima parte de un segundo, aunque en la mayoría de los casos no es necesario ser tan preciso).

var time1 = LocalTime.of(6, 15); // Hora y minutos
var time2 = LocalTime.of(6, 15, 30); // Hora, minutos y segundos
var time3 = LocalTime.of(6, 15, 30, 200); // Hora, minutos, segundos y nanosegundos

Estos tres objetos LocalTime representan diferentes momentos, pero todos están dentro del mismo minuto.

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)

Puedes combinar una fecha y una hora en un solo objeto LocalDateTime:

var dateTime1 = LocalDateTime.of(2022, Month.JANUARY, 20, 6, 15, 30);
var dateTime2 = LocalDateTime.of(date1, time1);

En la primera línea, se crea un objeto LocalDateTime especificando directamente todos los componentes de la fecha y la hora. En la segunda línea, se combinan previamente un objeto LocalDate (fecha) y un objeto LocalTime (hora) para crear el mismo LocalDateTime.

Hay muchas firmas de métodos porque hay más combinaciones. Las siguientes firmas de métodos usan valores enteros:

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)

Otros toman una referencia de Month:

```
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)
```

Finalmente, uno toma un LocalDate y LocalTime existentes:

```
public static LocalDateTime of(LocalDate date, LocalTime time)
```

Para crear un ZonedDateTime, primero necesitamos obtener la zona horaria deseada. Usaremos US/Eastern en nuestros ejemplos:

```
var zone = ZoneId.of("US/Eastern");
var zoned1 = ZonedDateTime.of(2022, 1, 20, 6, 15, 30, 200, zone);
var zoned2 = ZonedDateTime.of(date1, time1, zone);
var zoned3 = ZonedDateTime.of(dateTime1, zone);
```

Comenzamos obteniendo el objeto de zona horaria. Luego usamos uno de los tres enfoques para crear el ZonedDateTime. El primero pasa todos los campos individualmente. No recomendamos este enfoque—hay demasiados números, y es difícil de leer. Un mejor enfoque es pasar un objeto LocalDate y un objeto LocalTime, o un objeto LocalDateTime.

Aunque hay otras formas de crear un ZonedDateTime, solo necesitas conocer tres para el examen:

```
public static ZonedDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanos, ZoneId zone)
public static ZonedDateTime of(LocalDate date, LocalTime time, ZoneId zone)
public static ZonedDateTime of(LocalDateTime dateTime, ZoneId zone)
```

Observa que no hay una opción para pasar el enum Month. Además, no usamos un constructor en ninguno de los ejemplos. Las clases de fecha y hora tienen constructores privados junto con métodos estáticos que devuelven instancias. Esto se conoce como el patrón de fábrica. Los creadores del examen pueden presentarte algo como esto:

```
var d = new LocalDate(); // NO COMPILA
```

No caigas en esta trampa. No se te permite construir un objeto de fecha u hora directamente.

Otro truco es lo que sucede cuando pasas números no válidos a of(), por ejemplo:

```
var d = LocalDate.of(2022, Month.JANUARY, 32); // DateTimeException
```

No necesitas saber la excepción exacta que se lanza, pero es una clara:

```
java.time.DateTimeException: Valor no válido para DayOfMonth (valores válidos 1-28/31): 32
```

Manipulación de Fechas y Horas

Agregar a una fecha es fácil. Las clases de fecha y hora son inmutables. Recuerda asignar los resultados de estos métodos a una variable de referencia para que no se pierdan.

```
12: var date = LocalDate.of(2022, Month.JANUARY, 20);
13: System.out.println(date); // 2022-01-20
14: date = date.plusDays(2);
15: System.out.println(date); // 2022-01-22
16: date = date.plusWeeks(1);
17: System.out.println(date); // 2022-01-29
18: date = date.plusMonths(1);
19: System.out.println(date); // 2022-02-28
20: date = date.plusYears(5);
21: System.out.println(date); // 2027-02-28
```

Este código es agradable porque hace exactamente lo que parece. Comenzamos con el 20 de enero de 2022. En la línea 14, le agregamos dos días y lo reasignamos a nuestra variable de referencia. En la línea 16, agregamos una semana. Este método nos permite escribir un código más limpio que plusDays(7). Ahora la fecha es el 29 de enero de 2022. En la línea 18, agregamos un mes. Esto nos llevaría al 29 de febrero de 2022. Sin embargo, 2022 no es un año bisiesto (2020 y 2024 sí lo son). Java es lo suficientemente inteligente como para darse cuenta de que el 29 de febrero de 2022 no existe, y nos da el 28 de febrero de 2022 en su lugar. Finalmente, la línea 20 agrega cinco años.

También hay métodos agradables y fáciles para retroceder en el tiempo. Esta vez, trabajemos con LocalDateTime:

```
22: var date = LocalDate.of(2024, Month.JANUARY, 20);
23: var time = LocalTime.of(5, 15);
24: var dateTime = LocalDateTime.of(date, time);
25: System.out.println(dateTime);  // 2024-01-20T05:15
26: dateTime = dateTime.minusDays(1);
27: System.out.println(dateTime);  // 2024-01-19T05:15
28: dateTime = dateTime.minusHours(10);
29: System.out.println(dateTime);  // 2024-01-18T19:15
30: dateTime = dateTime.minusSeconds(30);
31: System.out.println(dateTime);  // 2024-01-18T19:14:30
```

La línea 25 imprime la fecha original del 20 de enero de 2024, a las 5:15 a.m. La línea 26 resta un día completo, llevándonos al 19 de enero de 2024, a las 5:15 a.m. La línea 28 resta 10 horas, mostrando que la fecha cambiará si las horas lo provocan, ajustándose y llevándonos al 18 de enero de 2024, a las 19:15 (7:15 p.m.). Finalmente, la línea 30 resta 30 segundos. Puedes ver que, de repente, el valor de la salida comienza a mostrar los segundos. Java es lo suficientemente inteligente como para ocultar los segundos y nanosegundos cuando no los estamos usando.

```
var date = LocalDate.of(2024, Month.JANUARY, 20);
var time = LocalTime.of(5, 15);
var dateTime = LocalDateTime.of(date, time)
   .minusDays(1).minusHours(10).minusSeconds(30);
```

Cuando tienes muchas manipulaciones por hacer, este encadenamiento es útil. Hay dos formas en que los creadores del examen pueden intentar confundirte. ¿Qué crees que imprime esto?

```
var date = LocalDate.of(2024, Month.JANUARY, 20);
date.plusDays(10);
System.out.println(date);
```

```
var date = LocalDate.of(2024, Month.JANUARY, 20);
date.plusDays(10);
System.out.println(date);
```

Imprime el 20 de enero de 2024. Agregar 10 días fue inútil porque el programa ignoró el resultado. Siempre que veas tipos inmutables, presta atención a este comportamiento.

Asegúrate de que el valor de retorno de una llamada a método no sea ignorado. El examen también puede evaluar si recuerdas qué incluye cada uno de los objetos de fecha y hora. ¿Ves qué está mal aquí?

```java
var date = LocalDate.of(2024, Month.JANUARY, 20);
date = date.plusMinutes(1);    // NO COMPILA
```

LocalDate no contiene tiempo. Esto significa que no puedes añadir minutos a él. Esto puede ser engañoso en una secuencia encadenada de operaciones de suma/resta, así que asegúrate de conocer qué métodos en la Tabla 4.6 pueden ser llamados en qué tipos.

imagen05 -> metodos localdate, localtime y localdateTime

Trabajando con Periodos:

¡Ahora sabes lo suficiente para hacer algo divertido con las fechas! Nuestro zoológico realiza actividades de enriquecimiento animal para darles algo agradable que hacer. El encargado principal del zoológico ha decidido cambiar los juguetes cada mes. Este sistema continuará durante tres meses para ver cómo funciona.

```
public static void main(String[] args) {
   var start = LocalDate.of(2022, Month.JANUARY, 1);
   var end = LocalDate.of(2022, Month.MARCH, 30);
   performAnimalEnrichment(start, end);
}

private static void performAnimalEnrichment(LocalDate start, LocalDate end) {
   var upTo = start;
   while (upTo.isBefore(end)) { // verifica si todavía está antes de la fecha final
      System.out.println("dar un juguete nuevo: " + upTo);
      upTo = upTo.plusMonths(1); // añade un mes
   }
}
```

Este código funciona bien. Añade un mes a la fecha hasta que alcanza la fecha final.
El problema es que este método no puede reutilizarse. Nuestro encargado del zoológico quiere probar diferentes horarios para ver cuál funciona mejor.

Afortunadamente, Java tiene una clase Period que podemos usar. Este código hace lo mismo que el ejemplo anterior:

```
public static void main(String[] args) {
   var start = LocalDate.of(2022, Month.JANUARY, 1);
   var end = LocalDate.of(2022, Month.MARCH, 30);
   var period = Period.ofMonths(1); // crea un periodo
   performAnimalEnrichment(start, end, period);
}

private static void performAnimalEnrichment(LocalDate start, LocalDate end, Period period)
{ // usa el periodo genérico
   var upTo = start;
   while (upTo.isBefore(end)) {
      System.out.println("dar un juguete nuevo: " + upTo);
      upTo = upTo.plus(period); // añade el periodo
   }
}
```

El método puede añadir cualquier periodo de tiempo que se le pase. Esto nos permite reutilizar el mismo método para diferentes periodos de tiempo si el encargado del zoológico cambia de opinión.

```java
var annually = Period.ofYears(1);          // cada 1 año
var quarterly = Period.ofMonths(3);        // cada 3 meses
var everyThreeWeeks = Period.ofWeeks(3);   // cada 3 semanas
var everyOtherDay = Period.ofDays(2);      // cada 2 días
var everyYearAndAWeek = Period.of(1, 0, 7); // cada año y 7 días
```

Hay una peculiaridad: No puedes encadenar métodos al crear un Period. El siguiente código parece equivalente al ejemplo everyYearAndAWeek, pero no lo es. Solo se usa el último método porque los métodos Period.of son métodos estáticos.

```java
var wrong = Period.ofYears(1).ofWeeks(1); // cada semana
```

Este código engañoso es realmente como escribir:

```java
var wrong = Period.ofYears(1);
wrong = Period.ofWeeks(1);
```

¡Esto claramente no es lo que pretendías! Por eso el método of() te permite pasar el número de años, meses y días. Todos están incluidos en el mismo periodo. Recibirás una advertencia del compilador sobre esto. Las advertencias del compilador te indican que algo está mal o es sospechoso sin fallar la compilación.

El método of() solo acepta años, meses y días. La capacidad de usar otro método factory para pasar semanas es simplemente una conveniencia. Como podrías imaginar, el periodo real se almacena en términos de años, meses y días.

Cuando imprimes el valor, Java muestra cualquier parte que no sea cero usando el formato mostrado en la Figura 4.9.

```java
System.out.println(Period.of(1,2,3));  // Imprime: P1Y2M3D
```

Como puedes ver, la P siempre inicia el String para mostrar que es una medida de periodo. Luego vienen el número de años, número de meses y número de días. Si alguno de estos es cero, se omite.

¿Puedes deducir qué imprime esto?

```java
System.out.println(Period.ofMonths(3));
```

La salida es P3M. Recuerda que Java omite cualquier medida que sea cero.

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

```java
3| var date = LocalDate.of(2022, 1, 20);
4| var time = LocalTime.of(6, 15);
5| var dateTime = LocalDateTime.of(date, time);
6| var period = Period.ofMonths(1);
7| System.out.println(date.plus(period));      // 2022-02-20
8| System.out.println(dateTime.plus(period));  // 2022-02-20T06:15
9| System.out.println(time.plus(period));      // Exception
```

Las líneas 7 y 8 funcionan como se espera. Añaden un mes al 20 de enero de 2022, resultando en 20 de febrero de 2022. La primera solo tiene la fecha, y la segunda tiene tanto la fecha como la hora.

La línea 9 intenta añadir un mes a un objeto que solo tiene tiempo. Esto no funcionará. Java lanza una UnsupportedTemporalTypeException y se queja de que intentamos usar una unidad no soportada: Meses.

Como puedes ver, tienes que prestar atención al tipo de objetos de fecha y hora en cada lugar que los veas.

Trabajando con Duraciones

Probablemente hayas notado que un Period es un día o más de tiempo. También existe Duration, que está pensado para unidades más pequeñas de tiempo. Para Duration, puedes especificar el número de días, horas, minutos, segundos o nanosegundos. Y sí, podrías pasar 365 días para hacer un año, pero realmente no deberías - para eso existe Period.

Duration funciona aproximadamente de la misma manera que Period, excepto que se usa con objetos que tienen tiempo. Duration se muestra comenzando con PT, que puedes pensar como un periodo de tiempo. Una Duration se almacena en horas, minutos y segundos. El número de segundos incluye segundos fraccionarios.

Podemos crear una Duration usando diferentes granularidades:

```java
var daily = Duration.ofDays(1);             // PT24H
var hourly = Duration.ofHours(1);           // PT1H
var everyMinute = Duration.ofMinutes(1);    // PT1M
var everyTenSeconds = Duration.ofSeconds(10);// PT10S
var everyMilli = Duration.ofMillis(1);      // PT0.001S
var everyNano = Duration.ofNanos(1);        // PT0.000000001S
```

Duration no tiene un método factory que tome múltiples unidades como lo hace Period. Si quieres que algo ocurra cada hora y media, especificas 90 minutos.

Duration incluye otro método factory más genérico. Toma un número y un TemporalUnit. La idea es, por ejemplo, "5 segundos." Sin embargo, TemporalUnit es una interfaz. Por el momento, solo existe una implementación llamada ChronoUnit.

El ejemplo anterior podría reescribirse así:

```java
var daily = Duration.of(1, ChronoUnit.DAYS);
var hourly = Duration.of(1, ChronoUnit.HOURS);
var everyMinute = Duration.of(1, ChronoUnit.MINUTES);
var everyTenSeconds = Duration.of(10, ChronoUnit.SECONDS);
var everyMilli = Duration.of(1, ChronoUnit.MILLIS);
var everyNano = Duration.of(1, ChronoUnit.NANOS);
```

ChronoUnit también incluye algunas unidades convenientes como ChronoUnit.HALF_DAYS para representar 12 horas.

ChronoUnit para Diferencias

ChronoUnit es una excelente manera de determinar qué tan separados están dos valores Temporal. Temporal incluye LocalDate, LocalTime, y similares. ChronoUnit está en el paquete java.time.temporal.

```java
var one = LocalTime.of(5, 15);
var two = LocalTime.of(6, 30);
var date = LocalDate.of(2016, 1, 20);
System.out.println(ChronoUnit.HOURS.between(one, two));     // 1
System.out.println(ChronoUnit.MINUTES.between(one, two));   // 75
System.out.println(ChronoUnit.MINUTES.between(one, date));  // DateTimeException
```

El primer print muestra que between trunca en lugar de redondear. El segundo muestra lo fácil que es contar en diferentes unidades. Solo cambia el tipo de ChronoUnit. El último nos recuerda que Java lanzará una excepción si mezclamos lo que se puede hacer con objetos de fecha versus tiempo.

Alternativamente, puedes truncar cualquier objeto con un elemento de tiempo. Por ejemplo:

```java
LocalTime time = LocalTime.of(3,12,45);
System.out.println(time);                                   // 03:12:45
LocalTime truncated = time.truncatedTo(ChronoUnit.MINUTES);
System.out.println(truncated);                              // 03:12
```

Este ejemplo pone en cero cualquier campo más pequeño que minutos. En nuestro caso, elimina los segundos.

"Usar una Duration funciona de la misma manera que usar un Period. Por ejemplo:

```
7: var date = LocalDate.of(2022, 1, 20);
8: var time = LocalTime.of(6, 15);
9: var dateTime = LocalDateTime.of(date, time);
10: var duration = Duration.ofHours(6);
11: System.out.println(dateTime.plus(duration)); // 2022-01-20T12:15
12: System.out.println(time.plus(duration)); // 12:15
13: System.out.println(date.plus(duration)); // UnsupportedTemporalTypeException
```

La línea 11 muestra que podemos agregar horas a un LocalDateTime, ya que contiene una hora. La línea 12 también funciona, ya que solo tenemos una hora. La línea 13 falla porque no podemos agregar horas a un objeto que no contiene una hora.

Probemos nuevamente, pero agregando 23 horas esta vez.

```
7: var date = LocalDate.of(2022, 1, 20);
8: var time = LocalTime.of(6, 15);
9: var dateTime = LocalDateTime.of(date, time);
10: var duration = Duration.ofHours(23);
11: System.out.println(dateTime.plus(duration)); // 2022-01-21T05:15
12: System.out.println(time.plus(duration)); // 05:15
13: System.out.println(date.plus(duration)); // UnsupportedTemporalTypeException
```

Esta vez vemos que Java avanza más allá del final del día. La línea 11 pasa al siguiente día ya que pasamos de la medianoche. La línea 12 no tiene un día, por lo que la hora simplemente se ajusta, al igual que en un reloj real."

"Period vs. Duration

Recuerda que Period y Duration no son equivalentes. Este ejemplo muestra un Period y un Duration de la misma longitud:

```
var date = LocalDate.of(2022, 5, 25);
var period = Period.ofDays(1);
var days = Duration.ofDays(1);

System.out.println(date.plus(period)); // 2022-05-26
System.out.println(date.plus(days));   // Unsupported unit: Seconds
```

Dado que estamos trabajando con un LocalDate, se requiere usar Period. Duration tiene unidades de tiempo, incluso si no las vemos, y están destinadas solo para objetos con tiempo.

imagen06 -> tabla de cuando usar Duration o Period

Trabajando con Instants:

"La clase Instant representa un momento específico en el tiempo en la zona horaria GMT.

Supongamos que quieres ejecutar un temporizador:

```
var now = Instant.now();

// hacer algo que tome tiempo

var later = Instant.now();
var duration = Duration.between(now, later);
System.out.println(duration.toMillis()); // Devuelve el número de milisegundos
```

En nuestro caso, lo que "tomaba tiempo" duró un poco más de un segundo, y el programa imprimió 1025.

Si tienes un ZonedDateTime, puedes convertirlo a un Instant:

```
var date = LocalDate.of(2022, 5, 25);
var time = LocalTime.of(11, 55, 00);
var zone = ZoneId.of("US/Eastern");
var zonedDateTime = ZonedDateTime.of(date, time, zone);
var instant = zonedDateTime.toInstant(); // 2022-05-25T15:55:00Z

System.out.println(zonedDateTime); // 2022-05-25T11:55-04:00[US/Eastern]
System.out.println(instant); // 2022-05-25T15:55:00Z
```

Las últimas dos líneas representan el mismo momento en el tiempo. El ZonedDateTime incluye una zona horaria. El Instant elimina la zona horaria y lo convierte en un instante de tiempo en GMT.

No puedes convertir un LocalDateTime a un Instant. Recuerda que un Instant es un punto en el tiempo. Un LocalDateTime no contiene una zona horaria y, por lo tanto, no es reconocido universalmente en todo el mundo como el mismo momento en el tiempo."

Teniendo en cuenta el horario de verano:

Algunos países observan el horario de verano. Esto significa que los relojes se ajustan una hora dos veces al año para aprovechar mejor la luz del sol. No todos los países participan, y aquellos que lo hacen usan diferentes fines de semana para el cambio. En el examen, solo debes trabajar con el horario de verano de Estados Unidos, que es lo que describimos aquí.

La pregunta te indicará si una fecha y hora mencionada cae en un fin de semana en el que se programa cambiar la hora. Si no se menciona en una pregunta, puedes asumir que es un fin de semana normal. El acto de avanzar o retrasar el reloj ocurre a las 2:00 a.m., que cae muy temprano en la mañana del domingo.

La Figura 4.10 muestra lo que sucede con los relojes. Cuando cambiamos nuestros relojes en marzo, el tiempo avanza de 1:59 a.m. a 3:00 a.m. Cuando cambiamos nuestros relojes en noviembre, el tiempo retrocede, y experimentamos la hora de 1:00 a.m. a 1:59 a.m. dos veces. Los niños aprenden esto como "adelantar el reloj en primavera y atrasarlo en otoño".

Por ejemplo, el 13 de marzo de 2022, avanzamos nuestros relojes una hora y saltamos de 2:00 a.m. a 3:00 a.m. Esto significa que no hay 2:30 a.m. ese día. Si quisiéramos saber la hora una hora después de las 1 1:30, sería 3:30.

```
var date = LocalDate.of(2022, Month.MARCH, 13);
var time = LocalTime.of(1, 30);
var zone = ZoneId.of("US/Eastern");
```

Este código en Java establece una fecha (13 de marzo de 2022), una hora (1:30 AM) y una zona horaria (Este de Estados Unidos) para representar un momento específico en el tiempo, teniendo en cuenta el cambio de horario.

```
var zoneId = ZoneId.of("US/Eastern");
var dateTime = ZonedDateTime.of(date, time, zoneId);
System.out.println(dateTime); // 2022-03-13T01:30-05:00[US/Eastern]
```

System.out.println(dateTime.getHour()); // 1
System.out.println(dateTime.getOffset()); // -05:00

dateTime = dateTime.plusHours(1);
System.out.println(dateTime); // 2022-03-13T03:30-04:00[US/Eastern]
System.out.println(dateTime.getHour()); // 3
System.out.println(dateTime.getOffset()); // -04:00

// ... (código similar para noviembre)

¡Absolutamente! Aquí tienes la traducción del código y una explicación detallada, teniendo en cuenta el contexto del horario de verano:

Traducción y Explicación
Código y Salida:

Java

var zoneId = ZoneId.of("US/Eastern");
var dateTime = ZonedDateTime.of(date, time, zoneId);
System.out.println(dateTime); // 2022-03-13T01:30-05:00[US/Eastern]
System.out.println(dateTime.getHour()); // 1
System.out.println(dateTime.getOffset()); // -05:00

dateTime = dateTime.plusHours(1);
System.out.println(dateTime); // 2022-03-13T03:30-04:00[US/Eastern]
System.out.println(dateTime.getHour()); // 3
System.out.println(dateTime.getOffset()); // -04:00

// ... (código similar para noviembre)
Explicación:

Este código de Java demuestra cómo el horario de verano afecta las representaciones de fecha y hora en la zona horaria de Estados Unidos (US/Eastern).

Establecimiento de la fecha y hora: Se crea un objeto ZonedDateTime para representar un momento específico (por ejemplo, el 13 de marzo de 2022 a la 1:30 AM) en la zona horaria de Estados Unidos.
Impresión de detalles: Se imprime la fecha y hora completa, la hora y el offset (diferencia horaria respecto a UTC).
Avanzando una hora: Se utiliza el método plusHours(1) para avanzar una hora.
Impresión de los nuevos detalles: Se vuelven a imprimir la fecha y hora, hora y offset para observar los cambios.
Observaciones clave:

Salto en la hora: Al avanzar una hora desde la 1:30 AM el 13 de marzo, se salta directamente a las 3:30 AM debido al cambio de horario.
Cambio en el offset: El offset (diferencia horaria respecto a UTC) también cambia de -05:00 a -04:00, reflejando el ajuste del horario de verano.
Comportamiento en noviembre: En noviembre, cuando se retrocede el reloj, se experimenta una hora duplicada. Por ejemplo, después de la 1:30 AM, la siguiente hora también es la 1:30 AM.
Manejo de horas inexistentes: Java maneja inteligentemente las horas que no existen debido al cambio de horario (como las 2:30 AM del 13 de marzo).
¿Qué significa esto?

Este código demuestra cómo Java maneja los cambios en el horario de verano, ajustando automáticamente la hora y el offset cuando se realizan operaciones con fechas y horas. Es importante tener en cuenta estos ajustes al trabajar con diferentes zonas horarias y períodos de tiempo.