Tarea Clase 5

1. Investigar qué diferencia tienen los tipos básicos de java con sus wrappers Ejemplo int Integer

boolean Boolean

long Long

float Float

**Tipos de datos primitivos en Java**

Java cuenta con un pequeño conjunto de tipos de **datos primitivos**. Podríamos considerarlos **fundamentales**, ya que la mayor parte de los demás tipos, los tipos estructurados o complejos, son composiciones a partir de estos más básicos. Estos tipos de datos primitivos sirven para gestionar los tipos de información más básicos, como números de diversas clases o datos de tipo verdadero/falso (también conocidos como "valores booleanos" o simplemente "booleanos").

De estos tipos primitivos, ocho en total, seis de ellos están destinados a facilitar el trabajo con números. Podemos agruparlos en **dos categorías**: tipos numéricos enteros y tipos numéricos en punto flotante. Los primeros permiten operar exclusivamente con números enteros, sin parte decimal, mientras que el segundo grupo contempla también números racionales o con parte decimal.

**Tipos numéricos enteros**

En Java existen **cuatro tipos** destinados a almacenar **números enteros**. La única diferencia entre ellos es el número de bytes usados para su almacenamiento y, en consecuencia, el rango de valores que es posible representar con ellos. Todos ellos emplean una representación que permite el almacenamiento de **números negativos y positivos**. El nombre y características de estos tipos son los siguientes:

**Tipos numéricos en punto flotante**

Los tipos numéricos en punto flotante permiten representar números tanto muy grandes como muy pequeños además de números decimales. Java dispone de 2 tipos concretos en esta categoría:

* float: conocido como tipo de **precisión simple**, emplea un total de 32 bits. Con este tipo de datos es posible representar números en el rango de 1.4x10-45 a 3.4028235x1038.
* double: sigue un esquema de almacenamiento similar al anterior, pero usando 64 bits en lugar de 32. Esto le permite representar valores en el rango de 4.9x10-324 a 1.7976931348623157x10308

Booleanos y caracteres

Aparte de los 6 tipos de datos que acabamos de ver, destinados a trabajar con números en distintos rangos, Java define otros dos tipos primitivos más:

boolean: tiene la finalidad de facilitar el trabajo con valores "verdadero/falso" (booleanos), resultantes por regla general de evaluar expresiones. Los dos valores posibles de este tipo son true y false.

char: se utiliza para almacenar caracteres individuales (letras, para entendernos). En realidad, está considerado también un tipo numérico, si bien su representación habitual es la del carácter cuyo código almacena. Utiliza 16 bits y se usa la codificación UTF-16 de Unicode.

Tipos de datos estructurados

Los tipos de datos primitivos que acabamos de ver se caracterizan por poder almacenar un único valor. Salvo este reducido conjunto de tipos de datos primitivos, que facilitan el trabajo con números, caracteres y valores booleanos, todos los demás tipos de Java son objetos, también llamados tipos estructurados o "Clases".

Los tipos de datos estructurados se denominan así porque en su mayor parte están destinados a contener múltiples valores de tipos más simples, primitivos. También se les llama muchas veces "tipos objeto" porque se usan para representar objetos. Puede que te suene más ese nombre.

Cadenas de caracteres

Aunque las cadenas de caracteres no son un tipo simple en Java, sino una instancia de la clase String, el lenguaje otorga un tratamiento bastante especial a este tipo de dato, lo cual provoca que, en ocasiones, nos parezca estar trabajando con un tipo primitivo.

Aunque cuando declaramos una cadena estamos creando un objeto, su declaración no se diferencia de la de una variable de tipo primitivo de las que acabamos de ver:

String nombreCurso = "Iniciación a Java";

Y esto puede confundir al principio. Recuerda: Las cadenas en Java son un objeto de la clase String, aunque se declaren de este modo.

Las cadenas de caracteres se delimitan entre comillas dobles, en lugar de simples como los caracteres individuales. En la declaración, sin embargo, no se indica explícitamente que se quiere crear un nuevo objeto de tipo String, esto es algo que infiere automáticamente el compilador.

Las cadenas, por tanto, son objetos que disponen de métodos que permiten operar sobre la información almacenada en dicha cadena. Así, encontraremos métodos para buscar una subcadena dentro de la cadena, sustituirla por otra, dividirla en varias cadenas atendiendo a un cierto separador, convertir a mayúsculas o minúsculas, etc.

Vectores o arrays

Los vectores son colecciones de datos de un mismo tipo. También son conocidos popularmente como arrays e incluso como "arreglos" (aunque se desaconseja esta última denominación por ser una mala adaptación del inglés).

Un vector es una estructura de datos en la que a cada elemento le corresponde una posición identificada por uno o más índices numéricos enteros.

También es habitual llamar matrices a los vectores que trabajan con dos dimensiones.

Los elementos de un vector o array se empiezan a numerar en el 0, y permiten gestionar desde una sola variable múltiples datos del mismo tipo.

Por ejemplo, si tenemos que almacenar una lista de 10 números enteros, declararíamos un vector de tamaño 10 y de tipo entero, y no tendríamos que declarar 10 variables separadas de tipo entero, una para cada número.

**Tipos definidos por el usuario**

Además de los tipos estructurados básicos que acabamos de ver (cadenas y vectores) en Java existen infinidad de clases en la plataforma, y de terceros, para realizar casi cualquier operación o tarea que se pueda ocurrir: leer y escribir archivos, enviar correos electrónicos, ejecutar otras aplicaciones o crear cadenas de texto más especializadas, entre un millón de cosas más.

Todas esas clases son tipos estructurados también.

Y por supuesto puedes crear tus propias clases para hacer todo tipo de tareas o almacenar información. Serían tipos estructurados definidos por el usuario.

**Tipos envoltorio o *wrapper***

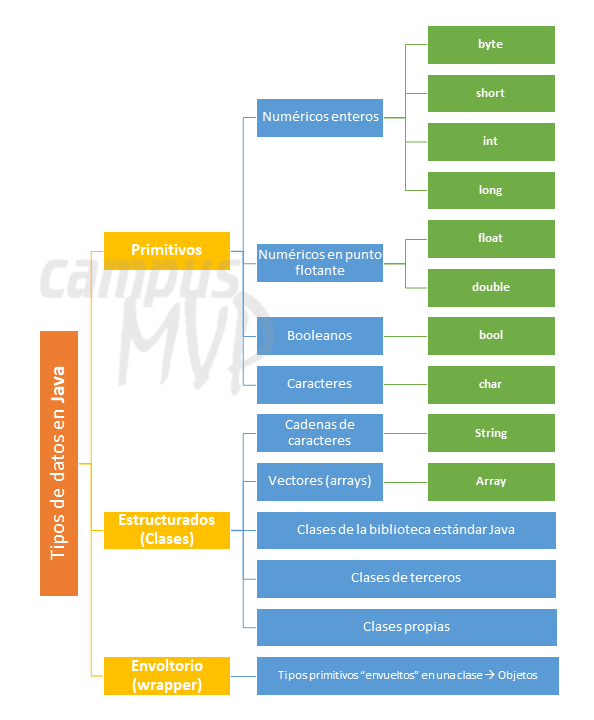
Java cuenta con tipos de datos estructurados equivalentes a cada uno de los tipos primitivos que hemos visto.

Así, por ejemplo, para representar un entero de 32 bits (int) de los que hemos visto al principio, Java define una clase llamada Integer que representa y "envuelve" al mismo dato, pero le añade ciertos métodos y propiedades útiles por encima.

Además, otra de las finalidades de estos tipos "envoltorio" es facilitar el uso de esta clase de valores allí donde se espera un dato por referencia (un objeto) en lugar de un dato por valor (para entender la diferencia entre tipos por valor y tipos por referencia lee **este artículo**. Aunque está escrito para C#, todo lo explicado es igualmente válido para Java).

Estos tipos equivalentes a los primitivos, pero en forma de objetos son: Byte, Short, Integer, Long, Float, Double, Boolean y Character (8 igualmente).

A continuación, te dejamos un esquema resumen de todo lo anterior para facilitarte que lo "digieras" viéndolo ya ordenado:



1. Investigar cómo es que puedo tener un método con el mismo nombre y diferentes argumentos.

**Sobrecarga** es la capacidad de un lenguaje de programación, que permite nombrar con el mismo identificador diferentes variables u operaciones.

En programación orientada a objetos la sobrecarga se refiere a la posibilidad de tener dos o más funciones con el mismo nombre, pero funcionalidad diferente. Es decir, dos o más funciones con el mismo nombre realizan acciones diferentes. El compilador usará una u otra dependiendo de los parámetros usados. A esto se llama también sobrecarga de funciones.

También existe la sobrecarga de operadores que al igual que con la sobrecarga de funciones se le da más de una implementación a un operador.

El mismo método dentro de una clase permite hacer cosas distintas en función de los parámetros.

Java no permite al programador implementar sus propios operadores sobrecargados, pero sí utilizar los predefinidos como el +. C++, por el contrario, si permite hacerlo.

Como es que puedo tener un constructor con mismo nombre y diferentes argumentos.

en Programación Orientada a Objetos (POO), un **constructor** es una subrutina cuya misión es inicializar un objeto de una clase. En el constructor se asignan los valores iniciales del nuevo objeto.

Se utiliza para crear tablas de clases virtuales y poder así desarrollar el polimorfismo, una de las herramientas de la programación orientada a objetos. Al utilizar un constructor, el compilador determina cuál de los objetos va a responder al mensaje (virtual) que hemos creado. Tiene un tipo de acceso, un nombre y un paréntesis.

En Java es un método especial dentro de una clase, que se llama automáticamente cada vez que se crea un objeto de esa clase.

Posee el mismo nombre de la clase a la cual pertenece y no puede devolver ningún valor (ni siquiera se puede especificar la palabra reservada void). Por ejemplo, si añadiéramos a la clase SSuma un constructor, tendríamos que llamarlo también SSuma. Cuando en una clase no se escribe propiamente un constructor, Java asume uno por defecto.

**Constructor por defecto**

Un constructor por defecto es un constructor sin parámetros que no hace nada. Sin embargo, será invocado cada vez que se construya un objeto sin especificar ningún argumento, en cuyo caso el objeto será iniciado con los valores predeterminados por el sistema (los atributos numéricos a cero, los alfanuméricos a nuloPeq, y las referencias a objetos a null).

1. Investigar que es static y que lo diferencia de no-static (no poner nada).

Un método estático pertenece a la clase en sí y un método no estático (también conocido como instancia) pertenece a cada objeto que se genera a partir de esa clase. Si su método hace algo que no depende de las características individuales de su clase, hágalo estático (hará que la huella del programa sea más pequeña). De lo contrario, debería ser no estático.

1. Investigar objetivo de los test unitarios y como se hacen desde la óptica de la teoría.

En programación, una **prueba unitaria** es una forma de comprobar el correcto funcionamiento de una unidad de código. Por ejemplo, en diseño estructurado o en diseño funcional una función o un procedimiento, en diseño orientado a objetos una clase. Esto sirve para asegurar que cada unidad funcione correctamente y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, verificamos que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve, que, si el estado inicial es válido, entonces el estado final es válido también.

La idea es escribir casos de prueba para cada función no trivial o método en el módulo, de forma que cada caso sea independiente del resto. Luego, con las Pruebas de Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

Características

Para que una prueba unitaria tenga la *calidad suficiente* se deben cumplir los siguientes requisitos:

* Automatizable: No debería requerirse una intervención manual. Esto es especialmente útil para integración continua.
* Completas: Deben cubrir la mayor cantidad de código.
* Repetibles o Reutilizables: No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. También es útil para integración continua.
* Independientes: La ejecución de una prueba no debe afectar a la ejecución de otra.
* Profesionales: Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Aunque estos requisitos no tienen que ser cumplidos al pie de la letra, se recomienda seguirlos o de lo contrario las pruebas pierden parte de su función.

Ventajas

El objetivo de las pruebas unitarias es aislar cada parte del programa y mostrar que las partes individuales son correctas. Proporcionan un contrato escrito que el trozo de código debe satisfacer. Estas pruebas aisladas proporcionan cinco ventajas básicas:

* Fomentan el cambio: Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura (lo que se ha dado en llamar refactorización), puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido defectos.
* Simplifica la integración: Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente. De esta manera se facilitan las pruebas de integración.
* Documenta el código: Las propias pruebas son documentación del código, puesto que ahí se puede ver cómo utilizarlo.
* Separación de la interfaz y la implementación: Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro, a veces usando objetos maquetados (mock object - maqueta) que habilitan de forma aislada (unitaria) el comportamiento de objetos complejos.
* Los errores están más acotados y son más fáciles de localizar: Dado que tenemos pruebas unitarias que pueden desenmascararlos.

Limitaciones

Es importante darse cuenta de que las pruebas unitarias no descubrirán todos los errores del código. Algunos enfoques se basan en la generación aleatoria de objetos para amplificar el alcance de las pruebas de unidad.[1](https://es.wikipedia.org/wiki/Prueba_unitaria#cite_note-1)​ Esta técnica se conoce como testing aleatorio (RT, por random testing). Por definición, sólo prueban las unidades por sí solas. Por lo tanto, no descubrirán errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto. Además, puede no ser trivial anticipar todos los casos especiales de entradas que puede recibir en realidad la unidad de programa bajo estudio. Las pruebas unitarias sólo son efectivas si se usan en conjunto con otras pruebas de software.

Regla de las tres A.

A la hora de definir código en el proceso de construcción de un desarrollo es necesario tener claros los siguientes objetivos:

* Realizar un código claro, eficiente y estructurado que facilite un posible mantenimiento del mismo.
* Producir un código con el menor número de errores posibles.
* Realizar un código en base a un estándar de construcción que facilite la reutilización de componentes en la construcción.
* Documentar con eficiencia el código para facilitar la compresión del mismo.
* Tener en cuenta las cuestiones de rendimiento a la hora de programar, intentando encontrar soluciones que minimicen el consumo de recursos del sistema.

1. Investigar que es el this.

**Palabra clave this**

Un método miembro de un objeto está asociado al objeto. Cuando este se está ejecutando podemos usar *this*, para conseguir una referencia al objeto asociado.

Dentro del método, podemos usar this.nombre para acceder al nombre del objeto asociado.

La palabra clave *this,* funciona igual dentro de un constructor.