





JAVA

Hola!

José Julián Ariza



Fundador de **Impacto Tecnológico** desde 2006.

Líder estratégico de Equipos de Desarrollo Ágil en modalidad Online y Presencial

Arquitecto de Software por naturaleza y DevOps por hobby

@JJArizaV – josejulian@impactotecnologico.net

@impactotecno



Temario

BLOQUE I

- Java 8 Platform Overview
- Java Syntax and Class Review
- Encapsulation and Subclassing
- Overriding methods,
Polymorphism and Static
Classes
- Abstract and Nested Classes
- Interfaces and Lambda
Expressions
- Collections and Generics
- Collections streams and filters

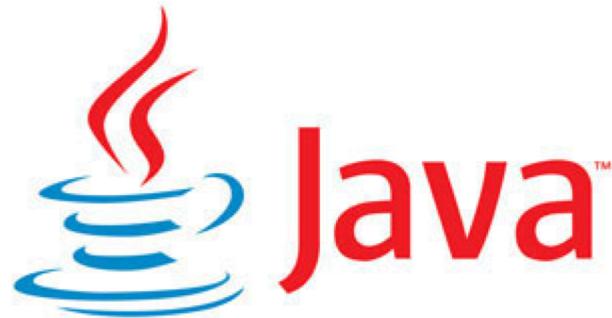
BLOQUE II

- Lambda Built-in Functional Interfaces
- Lambda Operations
- Exceptions and Assertions
- I/O Fundamentals
- Concurrency
- Database Application with JDBC

1. Visión General de JAVA 8

Hagamos una descripción general





Java

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principio de los años 90's.

Surgió como proyecto de la empresa como un lenguaje de programación sencillo y universal destinado a *electrodomésticos liderado por ...*



Visión General de Java 8

DATOS GENERALES

- Es la última versión del lenguaje de desarrollo
- Es el incremento de versión que añade funcionalidades nuevas al lenguaje de desarrollo
- Lanzado el 18 de Marzo de 2014

NUEVAS FUNCIONALIDADES

- Expresiones LAMBDA
- Referencias a Métodos
- Método por defecto
- Nuevas herramientas
- Stream API
- Date Time API
- Énfasis en las mejores prácticas para el manejo de valores null
- Nashorn

Características de Java

- Orientado a objetos:
 - Salvo los tipos de datos primitivos, todo es objeto en Java. Y además, Java se ha provisto de clases incorporadas que encapsulan los tipos primitivos.
- Sencillo
 - La sintaxis de Java es similar a la del lenguaje C y C++, pero evita características semánticas que les hacen complejos, confusos y no securizados
- Distribuido
 - Java implementa los protocolos de red estándares, lo que permite desarrollar aplicaciones cliente/servidor en arquitectura distribuida, con el fin de invocar tratamientos y/o recuperar datos de máquinas remotas.
- Interpretado
 - Un programa Java no lo ejecuta sino que lo interpreta la máquina virtual o JVM (Java Virtual Machine). Esto hace que sea más lento ...

```
1 // HelloWorld.java  
2 // The traditional first program for novice programmers!  
3 class HelloWorld {  
4     public static void main(String[] args) {  
5         System.out.println("Hello World!");  
6     }  
7 }
```

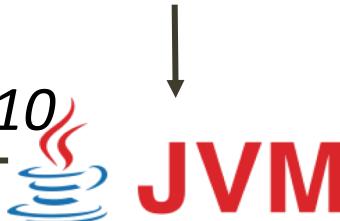


*javac
compilación*

bytecodes



010101010



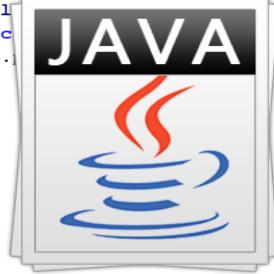
Características de Java

- ◆ Robusto
 - ◆ Java es un lenguaje fuertemente tipado y estricto. Por ejemplo, la declaración de las variables debe ser obligatoriamente explícita en Java. Se verifica el código (sintaxis, tipos) en el momento de la compilación y también de la ejecución, lo que permite reducir los errores y los problemas de incompatibilidad de versiones.
- ◆ Securizado
 - ◆ Dado los campos de aplicación de Java, es muy importante que haya un mecanismo que vigile la seguridad de las aplicaciones y los sistemas. El motor de ejecución de Java (JRE) es el encargado de esta tarea.
- ◆ Independiente de las arquitecturas
 - ◆ El compilador Java no produce un código específico para un tipo de arquitectura. De hecho, el compilador genera bytecode (lenguaje binario intermedio) que es independiente de cualquier arquitectura, de todo sistema operativo y de todo dispositivo de gestión de la interfaz gráfica de usuario (GUI).

Características de Java

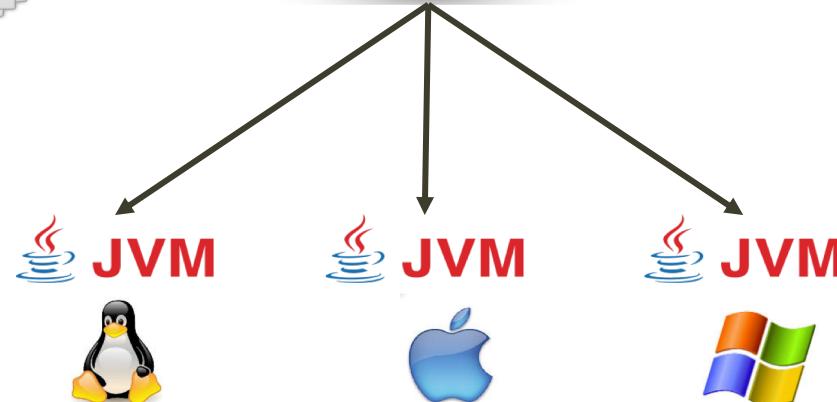
- ◆ **Portable**
 - ◆ Java es portable gracias a que se trata de un lenguaje interpretado.
- ◆ **Eficaz**
 - ◆ Incluso si un programa Java es interpretado, lo que es más lento que un programa nativo, Java pone en marcha un proceso de optimización de la interpretación del código, llamado JIT (Just In Time) o Hot Spot. Este proceso compila el bytecode Java en código nativo en tiempo de ejecución, lo que permite alcanzar el mismo rendimiento que un programa escrito en lenguaje C o C++.
- ◆ **Multihilo**
 - ◆ Java permite desarrollar aplicaciones que ponen en marcha la ejecución simultánea de varios hilos (o procesos ligeros). Esto permite efectuar simultáneamente varias tareas, con el fin de aumentar la velocidad de las aplicaciones

```
1 // HelloWorld.java  
2 // The traditional first program for novice programmers!  
3 class HelloWorld  
4     public static void main(String[] args) {  
5         System.out.println("Hello World!");  
6     }  
7 }
```



bytecode

javac



Ediciones Java

- Es el núcleo que contiene todas las herramientas necesarias para crear una aplicación java sencilla que puede ser un applet para navegador web o una aplicación desktop

Java Standard Edition



- Posee funcionalidades diseñadas para aplicaciones empresariales de mayor complejidad que incluye servicios web, XML,JMIS,RMI.

Java EE



- Proporciona un entorno robusto y flexible para las aplicaciones que se ejecutan en dispositivos embebidos y móviles en el Internet de las Cosas.

Java Embedded



- Creada para dispositivos de poca capacidad o potencia como teléfonos móviles o Smartphone, proporciona funcionalidades adicionales al core.

Java ME



Principales Diferencias con C y C++

Java	C++
Es tanto un lenguaje de programación como una plataforma de software.	Es sólo un lenguaje de programación.
Es un lenguaje puramente orientado a objetos.	Da soporte tanto a la programación estructurada como a la programación orientada a objetos.
Todas las declaraciones de variables y métodos deben estar dentro de la definición de la clase.	Las declaraciones de variables y funciones pueden estar presentes fuera de las definiciones de las clases. No es necesario para un programa en C++ tener una clase.
El lenguaje es independiente de la plataforma. El código Java, una vez escrito, puede ser ejecutado en cualquier plataforma.	El código C++, una vez escrito para una plataforma, necesita ser compilado de nuevo, y el código objeto reenlazado para ser ejecutado en otra plataforma diferente.
Maneja la memoria automáticamente.	Los programadores tienen que hacerse cargo de liberar la memoria no utilizada.

Principales Diferencias con C y C++

Java	C++
No se soporta características como sobrecarga de operadores y conversiones automáticas en ambos sentidos.	Da soporte a características como sobrecarga de operadores y conversiones automáticas en ambos sentidos.
Una clase no puede heredar directamente de más de una clase. Se da soporte a la herencia múltiple usando interfaces.	Una clase puede heredar directamente de más de una clase.
Tiene rutinas de librerías extensibles.	Sus rutinas de librerías no son extensibles.
La programación de redes es más fácil. Los objetos pueden ser accedidos a través de la red usando URLs.	La programación para redes es compleja, a menos que se usen APIs de terceros. C++, es un lenguaje, que no brinda soporte incorporado para programación de redes.
Los programadores no pueden usar apuntadores. Los apuntadores se usan internamente.	Los programadores pueden usar apuntadores.
Implementa arreglos verdaderos.	Se implementan los arreglos con aritmética de apuntadores.

Eclipse

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge y actualmente es desarrollado por la Fundación Eclipse, bajo licencia Software Libre.

Eclipse brinda soporte para varios lenguajes de programación como: c/C++, PHP, SED además de numerosos plugins para distintas tecnologías.

Eclipse está compuesto por lo siguiente:

- Plataforma principal - inicio de Eclipse, ejecución de plugins
- OSGi - una plataforma para bundling estándar.
- El Standard Widget Toolkit (SWT) - Un widget toolkit portable.
- JFace - manejo de archivos, manejo de texto, editores de texto
- El Workbench de Eclipse - vistas, editores, perspectivas, asistentes

2.

Sintaxis Java y Repaso del concepto de Clases



Java Sintaxis - Descripción

- Java es un lenguaje orientado a objetos
- El estilo de orientación a objeto es una abstracción del mundo que nos permite identificar objetos reales y modelarlos dentro del lenguaje para la resolución de problemas
- Todo lo que se maneja dentro del sistema se basa en la interrelación entre objetos y la forma de comunicarse unos con otros

Práctica de clases y objetos

- ◆ Indra quiere crear un software para gestión de aerolíneas, y las aerolíneas podrán ofrecer vuelos para distintos destinos.

Salida estándar en Java

Para la escritura en la consola se utiliza el método `System.out` para el flujo de salida "estándar".

Este flujo ya está abierto y listo para aceptar datos de salida.

Por lo general este flujo corresponde a mostrar la salida por consola o en otro destino de salida especificado por el entorno o el usuario.

Para aplicaciones simples Java, una forma típica de escribir una línea de datos de salida es:

```
System.out.println( "mensaje a mostrar"); // escribe y cambia de línea  
System.out.println(); //cambia de línea, se utiliza para dejar una línea en blanco  
System.out.print ("mensaje a mostrar"); // escribe y no cambia de línea
```

Tipos de Datos

Los tipos de datos primitivos se usan para definir variables en Java el cual posee ocho tipos de datos primitivos.

- Tipo primitivo
 - Tipo entero
 - Byte
 - Int
 - Short
 - Long
 - char
 - Tipo punto flotante
 - Float
 - double
 - Boolean

Tipos de Datos

Tipos de Datos Enteros

Los tipos de datos enteros se usan para almacenar valores enteros, existen cinco tipos diferentes de tipos de datos enteros, estos son: byte, short, int, long, y char.

El rango de valores que estos tipos de datos primitivos pueden almacenar se muestra a continuación:

byte	-2 ⁷ hasta 2 ⁷ - 1
short	-2 ¹⁵ hasta 2 ¹⁵ - 1
int	-2 ³¹ hasta 2 ³¹ - 1
long	-2 ⁶³ hasta 2 ⁶³ - 1

Las diferentes variables de los tipos de datos enteros se declaran como se muestra en el siguiente código:

```
byte b = 1;  
short s = 1;  
int i = 1;  
long l = 1;
```

Tipos de Datos

Tipos de Datos de Punto Flotante

Los tipos de datos de punto flotante se usan para almacenar valores de punto flotante. Los dos tipos de tipos de datos para punto flotante son float y double.

El rango de estos tipos de datos primitivos es el siguiente:

float	-3.4 * 10 ³⁸ hasta 3.4 * 10 ³⁸
double	-1.8 * 10 ³⁰⁸ hasta 1.8 * 10 ³⁰⁸

El siguiente código muestra la declaración de dos variables pertenecientes al tipo de datos de punto flotante:

```
float a = 1.1f;  
double d = 1.1;
```

Las variables literales float deben terminar con f o F.

Tipos de Datos

Tipo de Datos boolean

El tipo de datos boolean se usa para almacenar valores booleanos, es decir, true o false. La palabra clave boolean se usa para denotar el tipo de datos booleano.

El siguiente código muestra la declaración de una variable boolean:

Tipos de Datos

- La clase String soporta alguna sintaxis no estándar.
- Se puede instanciar un objeto String sin utilizar la palabra clave new; se prefiere esto:

```
String name1 = "Juan José";
```

- Se puede utilizar la palabra clave new, pero *no* se recomienda:

```
String name1 = new String("Córdoba");
```

- Un objeto String es inmutable; su valor no se puede cambiar.
- Un objeto String se puede utilizar con el símbolo del operador de concatenación de cadenas (+) para la concatenación.
- La concatenación crea una nueva cadena y la referencia

Comparativa entre Strings

Cuando se utiliza el operador `==` para comparar referencias de objetos con objetos `String`, el operador prueba si las direcciones de las referencias de objetos `String` de la memoria son iguales, no su contenido.

Para probar la igualdad entre cadenas de caracteres es necesario el uso del método `equals` de la clase `String`. La clase del ejemplo contiene dos nombres de empleados y un método para comparar los nombres

```
public class Employees {  
    public String name1 = "Juan José";  
    public String name2 = "Cordoba";  
    public void areNamesEqual() {  
        if (name1.equals(name2)) {  
            System.out.println("Same name.");  
        } else {  
            System.out.println("Different name.");  
        }  
    }  
}
```

StringBuilder

StringBuilder proporciona una alternativa variable a String, consiste en una clase normal Java con las siguientes características:

- ❑ Tiene un amplio juego de métodos para agregar, insertar y suprimir.
- ❑ Tiene muchos métodos para devolver una referencia al objeto actual.
- ❑ Se puede crear con la capacidad inicial que mejor se adapte a las necesidades.

Sin embargo String sigue siendo necesaria debido a que:

- ❑ Su uso puede ser más seguro que un objeto inmutable.
- ❑ Una clase de la API puede necesitar una cadena.
- ❑ Tiene muchos más métodos no disponibles en StringBuilder.

Variables

Variables Final

Las constantes en Java son definidas utilizando el modificador **final** que indica que el miembro tiene un valor constante.

- Características
 - Un miembro con un valor constante no puede ser modificado, si se trata de cambiar el valor se generará un error en tiempo de compilación.
 - Deben ser inicializadas con un valor en el momento de su declaración de lo contrario arrojará un mensaje de error, diciendo que la variable puede no haber sido inicializada.
 - El modificador **static** puede ser combinado con la variable final para declarar constantes de librerías. Esta combinación es particularmente útil al declarar constantes como: PI, E, etc.

El nombre de una variable final se representa en letras mayúsculas por convención tal como se muestra a continuación: **final float PI = 3.14;**

Operadores

En Java existen expresiones que hacen uso de varios operadores y son similares a expresiones aritméticas que usan muchos operadores.

Las variables usadas en estas expresiones se llaman *operandos*, existen 3 tipos de operandos:

- Los que solo necesitan un operando, son llamados operadores *unarios* (ejemplo: `++`).
- Los que requieren dos operandos se llaman operadores *binarios* (por ejemplo, `+`).
- Los que requieren tres operandos se llaman operadores *ternarios* (ejemplo: `?:?`).

A su vez Java soporta los siguientes tipos de operadores:

- OPERADORES
 - Operadores Aritméticos.
 - Operadores Relacionales.
 - Operadores Condicionales.
 - Operadores Ternarios.
 - Operadores de Bits.
 - Operadores de Asignación.

Operadores Aritméticos

OPERADOR	USO	DESCRIPCION
+	$a + b$	Permite adicionar valores almacenados en dos o más variables
-	$a - b$	Permite restar un valor almacenado en una variable de un valor almacenado en otra variable.
*	$a * b$	Permite multiplicar valores almacenados en dos variables.
/	a / b	Permite dividir un valor almacenado en una variable entre un valor almacenado en otra variable.
%	$a \% b$	Permite encontrar el residuo resultante de dividir un valor almacenado en una variable entre el valor almacenado en otra variable.

Precedencia de Operadores

Reglas de prioridad:

1. Operadores delimitados por un par de paréntesis.
2. Operadores de aumento y disminución.
3. Operadores de multiplicación y división, evaluados de izquierda a derecha.
4. Operadores de suma y resta, evaluados de izquierda a derecha.

Ejemplo:

$$c = 25 - ((5 * 4) / 2)) - 10 + 4;$$

El resultado real de la expresión cuando se evalúa según las reglas de prioridad, indicadas por los paréntesis: 9

Creación de construcciones if e if/else

Una sentencia if, o una construcción if, ejecuta un bloque de código si una expresión es *true*.

Sintaxis:

```
if (boolean_expression) {  
    ...code_block;  
} // end of if construct  
// program continues here
```

En donde:

- *boolean_expression* es una combinación de operadores relacionales, operadores condicionales y valores cuyo resultado es un valor true o false.
- *code_block* representa las líneas de código que se ejecutan si la expresión es true.

Condiciones Switch

switch indica una sentencia switch

case indica un valor que está probando

```
switch (variable) {  
    case literal_value:  
        <code_block>  
        [break;]  
    case another_literal_value:  
        <code_block>  
        [  
            break;  
        ]  
    default:  
        <code_block>  
}
```

variable es la variable cuyo valor desea probar

literal_value es cualquier valor válido que puede contener una variable

break es una palabra clave opcional que hace que el flujo de código salga inmediatamente de la sentencia switch

Declaración, instanciación e inicialización de una matriz unidimensional

- ❑ Una matriz contiene una serie de variables que puede ser cero a N.
- ❑ Las variables contenidas en la matriz no tienen nombre y son consideradas componentes de la matriz.
- ❑ Si una matriz tiene n componentes, los componentes de la matriz se referencian usando índices enteros de 0 a n - 1, inclusive.

Sintaxis:

```
tipo identificador;
```

En donde:

tipo es el tipo de dato que contendrá la matriz .

identificador es el nombre de la matriz.

Recorriendo una matriz

```
static float media (float datos[]) {  
    int i;  
    int n = datos.length;  
    float suma = 0;  
    for (i=0; i<n; i++)  
        suma = suma + datos[i];  
    return suma/n;  
}
```

Ciclos While

La sentencia while ejecuta continuamente un bloque de instrucciones mientras una condición particular, sea verdadera.

Sintaxis:

while (expression) { statement(s) }

Ejemplo:

```
class WhileDemo {  
    public static void main(String[] args){  
        int count = 1;  
        while (count < 11) {  
  
            System.out.println("Count is: " + count);  
            count++;  
        }  
    }  
}
```

Objetos



Java Sintaxis - Qué es un objeto

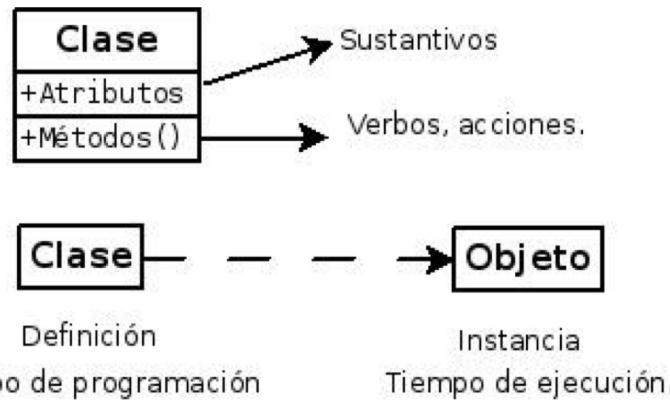
- A efectos de desarrollo de sistemas, un objeto es una entidad de programa que contiene dos características fundamentales: estado y comportamiento
- Los estados se almacenan en los campos, atributos o propiedades
- El comportamiento se define a través de los métodos. Los métodos actúan sobre los atributos internos el objeto y proveen los mecanismos de comunicación de objeto a objeto.

Java Sintaxis - ¿Y para qué necesitamos las clases?

- Los objetos son los elementos básicos para la programación empleando el paradigma OOP
- Las clases son los modelos o patrones a partir de las cuales se crean los objetos
- Los estados se pueden definir como tres tipos de posibles atributos: variables locales, variables de instancia y variables de clases
- Una clase puede definir el comportamiento a través de cualquier número de métodos que pueden acceder a cualquier tipo de valor de las variables.

Clases

- ◆ Una clase es un prototipo que se usa para definir las características y los comportamientos que son comunes a todos los objetos de un mismo tipo.



Clases

```
public class Persona {
```

Declaración de la
clase

```
public double altura =  
    1.70
```

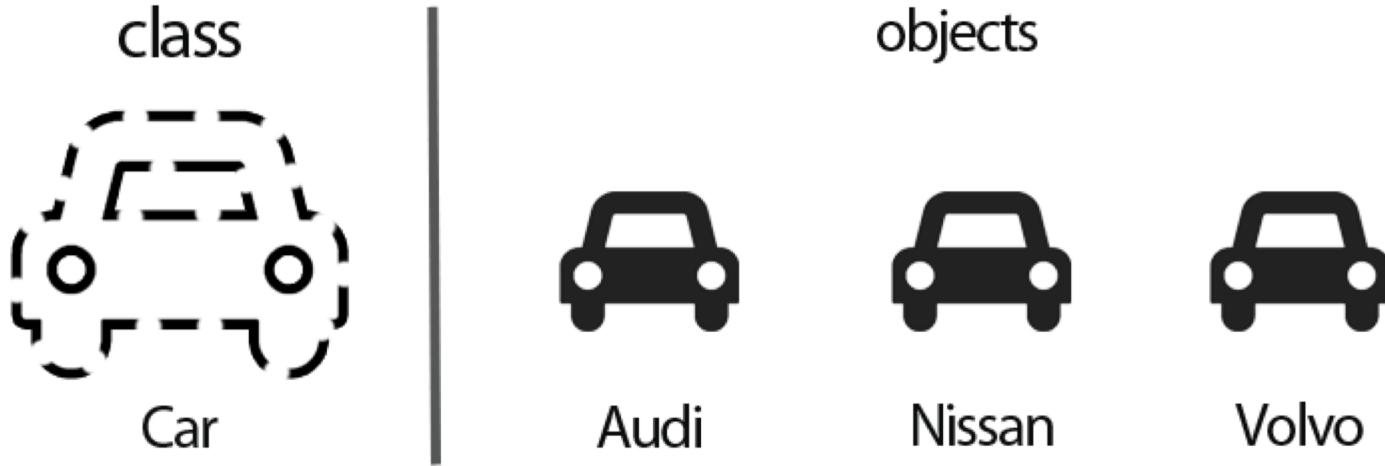
Declaración de
campo

```
public void respirar() {  
} // fin del método respirar
```

Método

Comentario

Java Sintaxis - ¿Y cómo vemos las clases?



Java Sintaxis - ¿Qué es un constructor?

- Una clase especial de método perteneciente a cada clase. Cada clase tiene al menos un constructor.
- Si no definimos un constructor, el compilador de Java construye un constructor por defecto que inicializa todas las variables a cero.
- Al momento de crear un objeto a partir de la clase al menos un constructor es invocado.
- La regla principal de un constructor es que debe tener el mismo nombre de la clase.
- Una clase puede tener más de un constructor.
- Los constructores se emplean de típicamente para inicializar variables de instancia o efectuar cualquier operación de inicio del objeto.

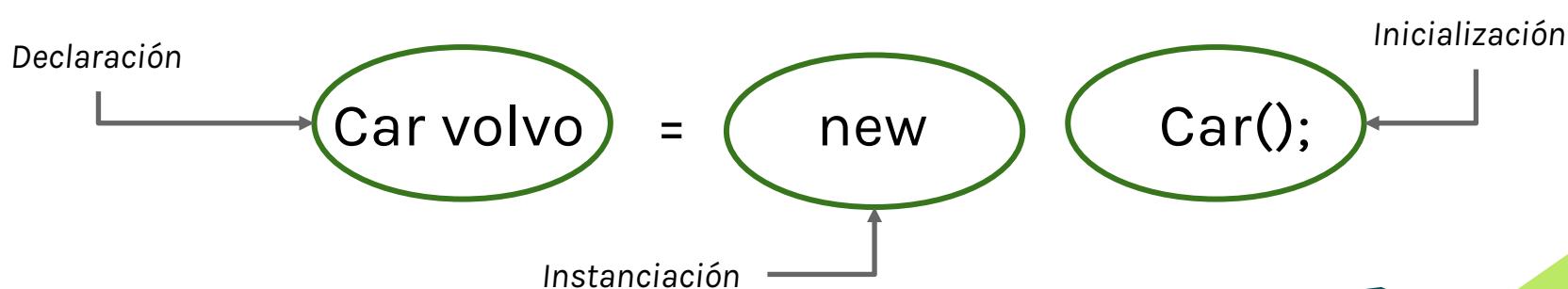
Identificación de Objetos

- ◆ Los objetos pueden ser físicos o conceptuales.
- ◆ Los objetos tienen características como nombre, tipo, tamaño, forma estas características son llamadas **atributos**.
- ◆ Los objetos pueden realizar acciones como definir un valor, visualizar una pantalla, etc. Estas acciones son llamados **métodos**.
- ◆ El valor de todos los atributos de un objeto se suele denominar **estado actual** del objeto

Java Sintaxis - ¿Cómo creamos un objeto?

Se requieren tres elementos para crear un objeto a partir de una clase:

- **Declaración:** Una declaración de una variable + el tipo de objeto que se desea crear.
- **Instanciación:** La palabra clave ‘new’ se emplea para crear el objeto.
- **Inicialización:** La palabra clave ‘new’ seguida de una llamada a un constructor realiza la inicialización del objeto de la clase seleccionada.



Reconocimiento de los criterios para definir objetos

Existen 2 criterios importantes para reconocer objetos:

1. Importancia del dominio de problemas:

- ¿Existe el objeto en los límites del dominio de problemas?
- ¿Es necesario el objeto para que se termine la solución?
- ¿Es necesario el objeto como parte de una interacción entre un usuario y el sistema?

2. Existencia independiente

Para que un elemento sea un objeto y no un atributo de otro objeto, debe existir independientemente en el contexto del dominio de problemas.

Java Sintaxis - ¿Cómo utilizamos los recursos del objeto?

Para acceder a las variable de instancia y a los métodos disponibles lo realizamos a través del objeto creado a partir de una clase:

- **Creamos un objeto:** NombreClase objeto = new ConstructorDeClase();
- **Podemos acceder a una variable de instancia de la siguiente forma:**
objeto.nombreVariableInstancia;
- **Podemos acceder a un método de la siguiente forma:**
objeto.nombreDelMetodo();

Práctica Individual

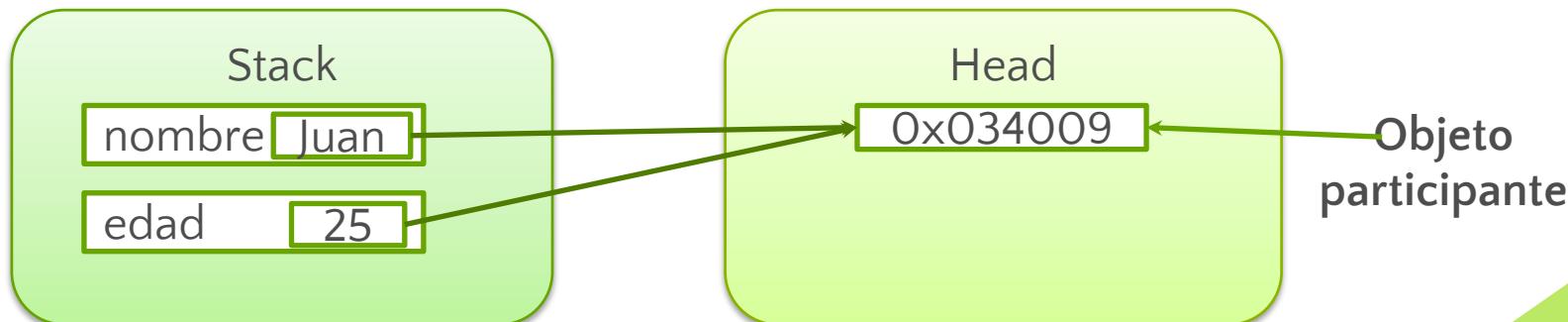
Se requiere el diseño de un aplicativo Java que permita realizar las operaciones matemáticas básicas a saber : suma, resta, multiplicación y división.

Define lo siguiente:

- Clases.
- Atributos.
- Métodos
- Comentarios

Almacenamiento de objetos en memoria

- En Java, la memoria se asigna cuando un objeto es creado a partir de una clase. Java da soporte a un manejo de memoria automático. Esto significa que la memoria asignada a un objeto está disponible para este, mientras el objeto esté siendo usado en el programa Java. Después que el objeto se usa en un programa, el recolector de basura(Garbage Collector) lo reclama y la memoria asignada a éste es liberada.



La sentencia import indica al compilador dónde están ubicadas las clases que estamos utilizando.

Para importar sólo una clase de un paquete:

```
import <nombre.paquete>.<NombreClase>;
```

Para importar todas las clases de un paquete:

```
import <nombre.paquete>.*;
```

El compilador añade a todos los ficheros la línea

```
import java.lang.*;
```

El paquete que contiene las clases fundamentales para programar en Java (System, String, Object...)

Métodos Estáticos

- ❑ Los métodos estáticos son definidos utilizando el modificador static, definiendo así un método de clase.
- ❑ Para acceder a un método de clase, no se necesita crear una instancia de la clase.
- ❑ Un método de clase no puede acceder a variables de instancia o métodos de instancia.
- ❑ Acceder a variables o métodos no estáticos desde un método estático genera un error en tiempo de compilación, aunque un método de clase puede ser accedido a través de una instancia.

Ejemplo:

```
static public void agregar(int j, int k) {  
    n = j + k;  
    System.out.println(n);  
}
```

Variables Estáticas

Una variable estática es una variable definida para la clase (compartida entre todas las instancias de una clase usando el modificador static..

Ejemplo:

```
public class Error{  
    int x ;  
    public static y ;  
  
    public static void main (String args[])  
    {  
        y = 15 ;  
        x = 20 ; //Error  
    }  
}
```

3. Encapsulamiento y Herencia (Subclassing)



Encapsulamiento y Subclases - Encapsulamiento

- Existen 4 principios fundamentales de la POO (OOP por sus siglas en inglés): **encapsulamiento, herencia (subclassing), polimorfismo y abstracción.**
- El **encapsulamiento** en Java es el mecanismo mediante el cual se empaquetan los datos (variables) y el código que actúa sobre los mismos (métodos) en una misma unidad funcional.
- Empleando este principio se colocan las variables dentro de una clase “escondidas” de las demás clases que solo podrán acceder a sus valores a través de métodos existentes en la misma clase, específicamente programados a tal fin.

Modificadores de Acceso

Se pueden usar los modificadores de acceso de Java para proteger las variables y métodos de una clase de ser accedidos sin garantía por parte de otras clases.

Los modificadores de acceso permitidos por Java son:

Modificadores de Acceso

Public: Accesibles en cualquier lugar desde donde la clase es accesible. Así mismo, son heredados por las subclases.

Protected: accesibles en las subclases y en el código del mismo paquete.

Package: Accesible en la clase y el paquete.

Modificadores de Acceso

Modificador	Clase	Subclase mismo paquete	Clase mismo Paquete	Subclase otro paquete	Clase otro paquete
private	si	no	no	no	no
protected	si	si	si	si	no
public	si	si	si	si	si
package	si	si	si	no	no

Encapsulamiento y Subclases - ¿Cómo se logra el encapsulamiento?

Siguiendo dos sencillos pasos:

1. Declarando todas las variables de una clase como privadas (private)
2. Creando dos tipos de métodos conocidos como getter and setter para cada variable, que son los únicos que podrán leer o escribir sobre dichas variables

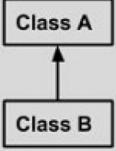
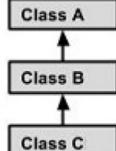
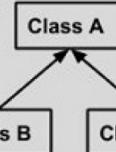
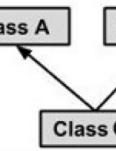
Encapsulamiento y Subclases - ¿Qué beneficios brinda el encapsulamiento?

- Para efectos de seguridad pudiéramos definir las variables de solo-lectura o solo- escritura.
- Una clase tiene el control total sobre la forma y manera en que se accede a los datos contenidos en ella.

Encapsulamiento y Subclases - Herencia o Subclassing

- La **herencia** es el proceso mediante el cual una clase adquiere (“hereda”) los atributos y métodos de otra clase. Este proceso permite el ordenamiento jerárquico de la información.
- La clase que hereda propiedades de otra clase se le llama subclase, aquella de la que se heredan las propiedades se le llama superclase.
- Palabras clave: **extends, implements, super, instanceof**
- Hay que hacer notar que Java NO SOPORTA la herencia múltiple (una clase heredando atributos de varias superclases) pero sí permite el mecanismo de implementar estructuras desde varias interfaces (lo cual le permite simular un tipo de herencia múltiple).

Encapsulamiento y Subclases - Tipos de Herencia

Single Inheritance		public class A { } public class B extends A { }
Multi Level Inheritance		public class A { } public class B extends A { } public class C extends B { }
Hierarchical Inheritance		public class A { } public class B extends A { } public class C extends A { }
Multiple Inheritance		public class A { } public class B { } public class C extends A,B { } } // Java does not support multiple inheritance

Práctica

A continuación los pasos a seguir:

-

Crear una superclase “Persona” que tenga los atributos generales de una persona y que “Pasajero” herede de persona añadiendo los atributos propios del pasajero (Ej: vuelos).

•Añadir en aerolínea métodos sobrecargados para consultarVuelos que pueda recibir:

1. solo origen
2. origen y destino

Sustitución de métodos

La sustitución de métodos ocurre cuando una subclases implementa métodos que ya tienen implantaciones en la superclase.

En este caso, las implantaciones de método en la subclase sustituyen la implantación de método de la superclase.

La sustitución de métodos se puede dar en los siguientes casos:

- Puede que los métodos que existen en la superclase:
 - No estén implantados en la subclase.
 - El método declarado en la superclase se utiliza en tiempo de ejecución.
 - Estén implantados en la subclase.
 - El método declarado en la subclase se utiliza en tiempo de ejecución.

Sobrecarga de constructores

- Java permite tener tantos constructores como se requiera para una clase, la única diferencia será en el número o el tipo de argumentos para una clase.
- Esto se denomina *sobrecarga del constructor*.
- Los constructores sobrecargados tienen un número diferente de parámetros o diferentes tipos de datos en los parámetros.
- Ésta es una forma de lograr polimorfismo en Java.

```
public Estudiante(){  
    public Estudiante(int nuevoId){  
        id = nuevoId;  
    }  
    public Estudiante(int nuevoId, String nuevoNombre){ //constructor  
sobrecargado  
        id = nuevoId;  
        nombre = nuevoNombre;  
    }  
}
```

Operador instanceof

El operador instanceof es una expresión cuyo tipo es un tipo de referencia y se utiliza para saber si la clase del objeto al que hace referencia el valor de la expresión en tiempo de ejecución es la asignación compatible con algún otro tipo de referencia.

El operador instanceof devuelve true o false basado en si el operando izquierdo es una instancia del operando derecho.

Sintaxis:

(Object reference variable) instanceof (class/interface type)

4.

Sobreescritura de Métodos, Polimorfismo y Clases Estáticas



Sobreescritura de Métodos, Polimorfismo y Clases Estáticas

- Sobreescritura de Métodos

- La sobreescritura de métodos se refiere a la posibilidad de redefinir el comportamiento de un método heredado desde una clase padre.
- Reglas:
 - ✓ Debe tener el mismo nombre, el mismo número de argumentos y tipo del valor de retorno del método original (o al menos un subtipo).
 - ✓ No puede tener un acceso más restrictivo que el original.
 - ✓ No se pueden sobreescribir tipos final o static.
 - ✓ No es obligado pero sí recomendable colocar la anotación “@override” para que el compilador identifique qué es lo que estamos haciendo y nos ayude a valorar errores.

Sobreescritura de Métodos, Polimorfismo y Clases Estáticas

- ¿Es la **sobreescritura** lo mismo que la **sobrecarga**?

- La sobrecarga de métodos se refiere a la posibilidad de generar métodos con el mismo nombre pero con una firma (juego de parámetros) distintos.
- Reglas:
 - ✓ No pueden haber dos métodos con el mismo nombre y los mismos parámetros.
 - ✓ Pueden existir n-número de métodos siempre y cuando tengan diferentes parámetros.
- Ejemplo: clase motor, método ignición. Pueden existir métodos sobrecargados para ignición dependiendo del tipo de motor: eléctrico, gasolina, diesel, etc.

Sobrecarga

```
public class DataArtist {  
    ...  
    public void draw(String s) {  
        ...  
    }  
    public void draw(int i) {  
        ...  
    }  
    public void draw(double f) {  
        ...  
    }  
}
```

Sobreescritura de Métodos, Polimorfismo y Clases Estáticas

- Polimorfismo

- El polimorfismo se refiere a la habilidad de los objetos de manifestar o “comportarse” como otros objetos. Dentro de la OOP usualmente se da cuando empleamos referencias a un objeto padre para referirnos a un objeto hijo.
- Reglas:
 - ✓ En Java una misma variable referenciada puede hacer referencia a más de un tipo de clase.
 - ✓ El conjunto de las que pueden ser referenciadas está restringido por la herencia o la implementación (si el tipo de clase hereda o implementa el tipo de la variable referenciada)

Sobreescritura de Métodos, Polimorfismo y Clases Estáticas

- Clases Estáticas

- Las clases estáticas son una forma de agrupar clases en Java.
- Java no permite crear una clase de nivel superior como estática.
- Solamente clases internas a una clase superior pueden ser definidas como estáticas.
- La creación de variables del tipo de una clase estática puede ser independiente de la creación de la clase de nivel superior que la contiene.
- Las clases internas estáticas pueden acceder a atributos estáticos de la clase que la contiene. NO podrá acceder a métodos o atributos que no sean estáticos.
- Por tanto, las clases internas NO estáticas difieren de las estáticas, en que mientras una clase estática puede instanciarse independientemente de la clase que la contiene, aquellas NO estáticas pertenecen y requieren la creación de un objeto de la clase contenedora.

Práctica

- ❑ Crear una clase Área que contenga métodos estáticos sobrecargados con el nombre area() que pueden ser usados para calcular el área de un círculo, triángulo, rectángulo y un cilindro.
- ❑ Cree una clase Usuario que utilice los métodos de Área para calcular el área de diferentes figuras geométricas y la imprime en la salida est\'andar.

Clases anidadas

- Una clase anidada es una clase cuya definición está dentro de otra clase.
- Si una clase contiene a una clase anidada, es conocida como *clase exterior*.
- Estas clases son consideradas miembro de la clase contenedora y pueden ser final o abstract.

Existen dos tipos de *Clases Anidadas*:

● **Clase Anidada Estática:** esta clase se declara como miembro estático (*static*) de la clase exterior y tiene todas las características de cualquier otro miembro estático.

Clase Interna: Esta clase se declara como miembro no-estático de la clase exterior y tienen todas las características de cualquier otro miembro de instancia. Una instancia de alguna clase interna sólo puede existir si una instancia de la clase contenedora existe. La clase interna no puede tener miembros estáticos propios.

Clases anidadas

- Las clases anidadas tienen acceso total a los miembros de su clase exterior, incluso a los miembros privados.
- Ahora como miembro también puede ser private, protected, además de public o package que es una posibilidad para cualquier clase.

```
class ClaseExterna{  
    //miembros de ClaseExterna  
    ...  
    public static class ClaseAnidadaEstatica {  
        //miembros de ClaseAnidadaEstatica  
        ...  
    }  
    private class ClaseInterna {  
        //miembros de ClaseInterna  
    }  
}
```

Métodos de argumentos variables

- A partir de Java 5 se permite especificar métodos que tengan un número variable de argumentos de un tipo especificado.
- Estos métodos se denominan varargs y pueden ser llamados con cero o más argumentos a la posición variable de argumentos.
- Un parámetro varargs es tratado como una matriz en el elemento que define.
- Si no sabemos cuántos argumentos tendremos que pasar en el método, varargs es el mejor enfoque.

Ejemplo:

Métodos de argumentos variables

```
class VarargsEjemplo{  
  
    static void display(String... values){  
        System.out.println("invocacion del metodo display");  
    }  
  
    public static void main(String args[]){  
  
        display(); //cero argumentos  
        display("mi","nombre","es","varargs"); //cuatro argumentos  
    }  
}
```

5.

Clases abstractas y anidadas



Clases abstractas y anidadas - Abstractas

- Una clase abstracta se parece en su diseño a una clase normal, pero sirve para servir de estructura padre a otras clases.
- Una clase abstracta se define con la palabra clave “abstract”.
- Características:
 - ✓ No puede ser instanciada (no se puede emplear “new”).
 - ✓ Si un método se considera o define como abstracto entonces la clase debe ser definida como abstracta.
 - ✓ Los métodos definidos como “abstract” no tienen cuerpo.
 - ✓ La clase que herede de una clase abstracta debe definir todos los métodos de la clase padre.

Clases Abstractas

- Las clases abstractas pueden tener dos tipos de métodos: métodos que tienen implementación y métodos que no tiene implementación.
- Los métodos implementados en una clase abstracta son métodos concretos.
- Los métodos que no tienen implementación son métodos abstractos.

Sintaxis:

```
abstract tipoRetorno nombreMetodo(argumentos);
```

Ejemplo:

```
abstract class U {  
    abstract void metodo1();  
    void metodo2() {  
        System.out.println("Dentro de metodo2 en la clase U");  
    }  
}
```

Comparativa

Clases Abstractas	Clases Concretas
Se usan para generalizar clases concretas.	Se usan para representar objetos del mundo real.
No pueden ser instanciadas.	Pueden ser instanciadas.
Definen métodos abstractos para los cuales la implementación no es provista.	No definen métodos abstractos.
Pueden o no proveer implementación para cualquiera de los métodos.	Deben proveer implementación para todos sus métodos.

Interfaces

- Las interfaces son similares a las clases, en términos de sintaxis, pero no soportan ninguna variable de instancia.
- Una interfaz especifica el prototipo o comportamiento de una clase.
- Todos los métodos declarados en una interfaz son métodos abstractos, es decir, no se les proporciona implementación.
- Una interfaz no puede ser instanciada.

Sintaxis:

```
modificadorDeAcceso interface nombreInterfaz {  
  
    tipoDeDatos nombreVar1 = valor;  
    tipoDeRetorno nombreMetodo1(listaDeArgumentos);  
  
}
```

Interfaces

Tanto las clases abstractas como las concretas pueden implementar una interfaz.

Una clase concreta que implemente la interfaz tiene que proveer la implementación para todos los métodos declarados en la interfaz.

La siguiente es la estructura general de la declaración de clase para la clase que implementa una interfaz:

```
modificadorDeAcceso class nombreClase [extends nombreSuperClase]  
[implements interfaz1, interfaz2, ...] {  
  
    // cuerpo de la clase
```

Manejo de Errores



Manejo de Excepciones

En java los errores son llamados excepciones y se encuentran de 2 tipos: excepciones verificadas y excepciones no verificadas.

Excepciones verificadas

- Son aquellas que el programador debe capturar y manejar dentro de la aplicación.
- Si el programador no captura una excepción verificada se producirá un error en tiempo de compilación.

Excepciones no verificadas

- Las excepciones no verificadas son excepciones del tiempo de ejecución, las cuales son detectadas por la JVM.
- Las excepciones no verificadas son lanzadas en un programa Java cuando hay un problema en la JVM.

Manejo de Excepciones

Java tiene incorporado la capacidad para asegurar que las excepciones son manejadas dentro del programa.

Para el manejo de excepciones se utilizan los siguientes recursos:

Manejo de errores

Bloques try y catch

Sentencia throw

Bloque finally

La Clase Throwable

Tipos de Excepciones

- ◆ Existen tres tipos de excepciones: errores, comprobadas (en adelante *checked*) y no comprobadas (en adelante *unchecked*).
 - ◆
 - ◆ Una excepción de tipo checked representa un error del cual técnicamente podemos recuperarnos. Por ejemplo, una operación de lectura/escritura en disco puede fallar porque el fichero no exista, porque este se encuentre bloqueado por otra aplicación, etc.
 - ◆
 - ◆ Todos estas situaciones, además de ser inherentes al propósito del código que las lanza (lectura/escritura en disco) son totalmente ajenas al propio código, y deben ser (y de hecho son) declaradas y manejadas mediante excepciones de tipo checked y sus mecanismos de control.

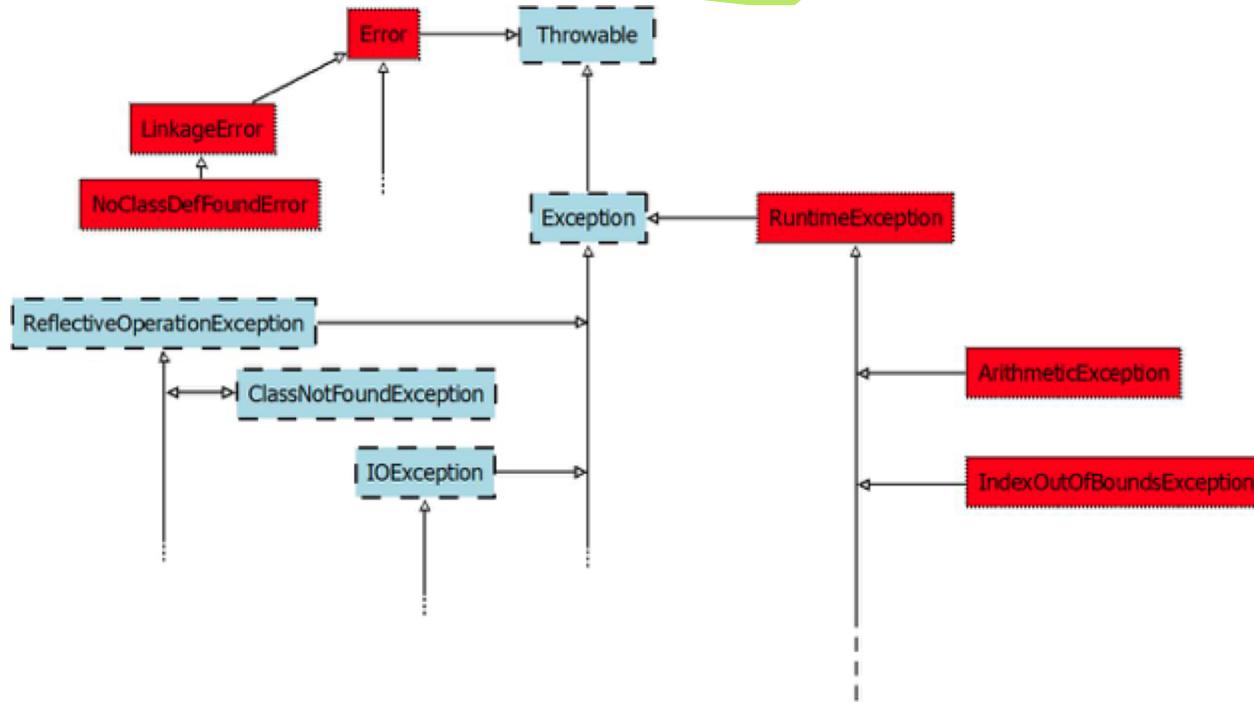
Tipos de Excepciones

- ◆ Una excepción de tipo unchecked representa un error de programación. Uno de los ejemplos más típicos es el de intentar leer en un array de N elementos un elemento que se encuentra en una posición mayor que N

```
int[] numerosPrimos = {1, 3, 5, 7, 9, 11, 13, 17, 19, 23};  
// Array de diez elementos  
  
int undecimoPrimo = numerosPrimos[10];  
// Accedemos al undécimo elemento mediante el literal numérico 10
```

- ◆ El aspecto más destacado de las excepciones de tipo unchecked es que no deben ser forzosamente declaradas ni capturadas (en otras palabras, no son comprobadas). Por ello no son necesarios bloques try-catch ni declarar formalmente en la firma del método el lanzamiento de excepciones de este tipo.

Tipos de Excepciones



Manejo de excepciones

- ◆ Para el manejo de excepciones los lenguajes proveen ciertas palabras reservadas, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones adicionales antes de interrumpir el programa.
- ◆ Lo más común es encontrar: try, catch y finally.
- ◆ Dentro del bloque try se ubica todo el código que pueda llegar a *levantar* una excepción, se utiliza el término *levantar* o *lanzar* para referirse a la acción de generar una excepción.

Manejo de excepciones

- ◆ A continuación se ubica el bloque catch, que se encarga de capturar la excepción y nos da la oportunidad de procesarla mostrando por ejemplo un mensaje adecuado al usuario.
- ◆ Dado que dentro de un mismo bloque try pueden producirse excepciones de distinto tipo, es posible utilizar varios bloques catch, cada uno para capturar un tipo distinto de excepción.
- ◆ Finalmente, puede ubicarse un bloque finally donde se escriben las sentencias de finalización, que son típicamente acciones de limpieza.
- ◆ La particularidad del bloque finally es que se ejecuta siempre, haya surgido una excepción o no.

Procesamiento y propagación de excepciones

- ◆ Vamos a ver qué se supone que hagamos al atrapar una excepción. En primer lugar podríamos ejecutar alguna lógica particular del caso como: cerrar un archivo, realizar una procesamiento alternativo al del bloque try, etc.
- ◆ Pero más allá de esto tenemos algunas opciones genéricas que consisten en: dejar constancia de la ocurrencia de la excepción, propagar la excepción o, incluso, hacer ambas cosas.
- ◆ Es posible, por otra parte, que luego de realizar algún procesamiento particular del caso se quiera que la excepción se propague hacia la función que había invocado a la función actual. Para hacer esto Python nos brinda la instrucción raise y en Java throw

Procesamiento y propagación de excepciones

- ◆ También podría ocurrir que en lugar de propagar la excepción tal cual fue atrapada, quisiéramos lanzar una excepción distinta, más significativa para quien invocó a la función actual y que posiblemente contenga cierta información de contexto.

Para levantar una excepción de cualquier tipo, utilizamos también la sentencia `throw`, pero indicándole el tipo de excepción que deseamos lanzar y pasando a la excepción los parámetros con información adicional que queramos brindar.

Práctica de Propagación de Excepciones

```
public static void main (String args[]) {  
    try {  
        createFile();  
    }  
    catch (IOException ioe) {  
        System.out.println(ioe);  
    }  
}  
  
public static void createFile() throws IOException {  
    File testF = new File("c:/notWriteableDir");  
    File tempF = testF.createTempFile("te", null, testF);  
    System.out.println("Temp filename is " +  
        tempF.getPath());  
    int myInt[] = new int[5];  
    myInt[5] = 25;
```

Procesamiento y propagación de excepciones

- ◆ Hay que tener presente que cuando se relanza una excepción estamos forzando al código cliente de nuestro método a capturarla o relanzarla. Una excepción que sea relanzada una y otra vez *hacia arriba* terminará llegando al método primigenio y, en caso de no ser capturada por éste, producirá la finalización de su hilo de ejecución (thread).
- ◆ La dos preguntas que debemos hacernos en este momento es:
 - ◆ ¿Cuándo capturar una excepción?
 - ◆ ¿Cuándo relanzarla?

La respuesta es muy simple.

Procesamiento y propagación de excepciones

- ◆ Capturamos una excepción cuando:

- ◆ Podemos recuperarnos del error y continuar con la ejecución

- ◆ Queremos registrar el error

- ◆ Queremos relanzar el error con un tipo de excepción distinto

- ◆ En definitiva, cuando tenemos que realizar algún tratamiento del propio error. Por contra, relanzamos una excepción cuando:

- ◆ No es competencia nuestra ningún tratamiento de ningún tipo sobre el error que se ha producido

Malas prácticas de uso

```
try { // Código que declara lanzar excepciones }
catch(Exception ex) {}
```

- ◆ El código anterior ignorará cualquier excepción que se lance dentro del bloque try, o mejor dicho, capturará toda excepción lanzada dentro del bloque try pero la silenciará no haciendo nada (frustrando así el principal propósito de la gestión de excepciones checked: *gestiónala o reláñzala*).
- ◆ Cualquier error de diseño, de programación o de funcionamiento en esas líneas de código pasará inadvertido tanto para el programador como para el usuario.

Malas prácticas de uso

- ◆ Otro abuso del mecanismo de tratamiento de excepciones es cuando se está intentando escribir código que mejore el rendimiento de la aplicación:

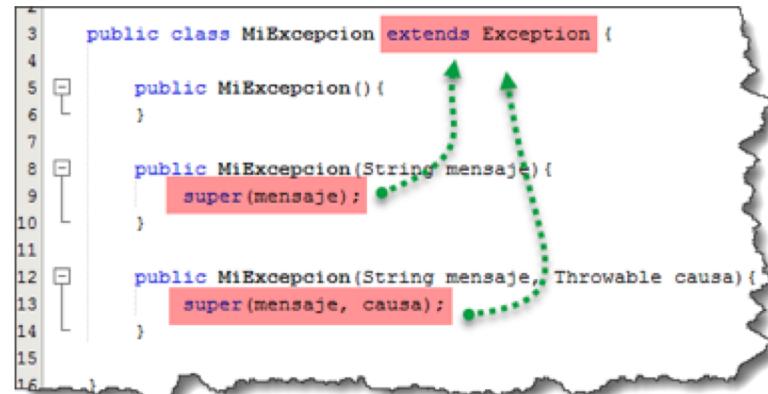
```
try {
    int i = 0;
    while(true) {
        System.out.println(numerosPrimos[i++]);
    }
} catch (ArrayIndexOutOfBoundsException aioobex)
{...}
```

- ◆ El ejemplo anterior itera el array de números primos sin preocuparse de los límites del array hasta sobrepasar el índice máximo, momento en el cual se lanzará una excepción de tipo `ArrayIndexOutOfBoundsException` que será capturada y silenciada.

Excepciones Propias

- ◆ Excepción definida por el usuario o excepción personalizada es crear su propia clase de excepción y arroja esa excepción usando la palabra clave 'throw'. Esto puede hacerse extendiendo la clase Exception.
- ◆ Para crear nuestra propia excepción, no necesitamos mucho.

```
3  public class MiExcepcion extends Exception {  
4  
5      public MiExcepcion(){  
6  }  
7  
8      public MiExcepcion(String mensaje){  
9          super(mensaje);  
10     }  
11  
12     public MiExcepcion(String mensaje, Throwable causa){  
13         super(mensaje, causa);  
14     }  
15  
16 }
```



Práctica de Excepciones Propias

- ◆ Crear excepciones propias para los casos
- ◆ Error de login
- ◆ Error de registro
- ◆ Error de reset de password
- ◆ Super clase para las 3 clases anteriores: GestiónSignUp
- ◆ Validar el tipo de excepción en un método main

Buenas prácticas de uso

- ◆ Una de las novedades que incorporó Java 7 es la sentencia *try-with-resources* con el objetivo de cerrar los recursos de forma automática en la sentencia *try-catch-finally* y hacer más simple el código.
- ◆ Aquellas variables cuyas clases implementan la interfaz AutoCloseable pueden declararse en el bloque de inicialización de la sentencia *try-with-resources* y sus métodos close() serán llamados después del bloque *finally* como si su código estuviese de forma explícita.

```
public static String readFirstLineFromFile(String path) throws IOException {  
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }  
}
```

Recomendaciones de uso

- ◆ Un buen uso del tratamiento de excepciones es usar excepciones que ya existen, en lugar de crear las tuyas propias, siempre que ambas fueran a cumplir el mismo cometido (que es básicamente informar y, en caso de las checked, obligar a gestionar). Se suelen usar excepciones que ya existen cuando se dispone de un profundo conocimiento del API que se está usando (en otras palabras, *experiencia*).
- ◆ Si un argumento pasado a uno de tus métodos no es del tipo esperado, o no tiene el formato correcto, primero revisa si puedes usar una excepción existente: `IllegalArgumentException` en lugar de crear tu propia excepción. Esto es bueno porque:
 - ◆ Uno de los pilares de Java es la reutilización de código (no reinventes la rueda)
 - ◆ Tu código es más universal (`FormatoInvalidoException` puede no significar nada para otra personas)

Función multi-catch

Excepciones multi-catch se han añadido a Java SE 7 para facilitar el manejo de excepciones de forma más fácil y más concisa, las mismas se agregan en un solo bloque catch y se separan con |:

```
try{  
    // Código que puede generar las excepciones  
}  
catch( ParseException | IOException exception){  
    // Código para manejo de la excepción  
}
```

Colecciones y Generics



Generics

- ◆ Los *generics* son importantes ya que permiten al compilador informar de muchos errores de compilación que hasta el momento solo se descubrirían en tiempo de ejecución
- ◆ Permiten eliminar los *cast* simplificando, reduciendo la repetición y aumentando la legibilidad el código.
- ◆ Los *generics* permiten usar tipos para parametrizar las clases, interfaces y métodos al definirlas.
- ◆ Los beneficios:
 - ◆ Comprobación de tipos más fuerte en tiempo de compilación.
 - ◆ Eliminación de *casts* aumentando la legibilidad del código.
 - ◆ Posibilidad de implementar algoritmos genéricos, con tipado seguro

Generics

```
public class Limite<T> {  
  
    private T t;  
  
    public T get() { return t; }  
  
    public void set(T t) { this.t = t; }  
  
}
```

```
Limite <Integer> caja1 = new Limite <Integer>();
```

```
Limite <Integer> caja2 = new Limite <>();
```

Práctica de Generics

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}
```

```
public class OrderedPair<K, V> implements Pair<K, V> {  
    private K key;  
    private V value;  
    public OrderedPair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
    public K getKey() { return key; }  
    public V getValue() { return value; }  
}
```

```
Pair<Integer, String> p1 = new OrderedPair<>(1, "rojo");  
Pair<Integer, String> p2 = new OrderedPair<>(2, "azul");
```

Generics

- ◆ Para nombrar Generics se acostumbra a usar:
 - ◆ E: elemento de una colección.
 - ◆ K: clave.
 - ◆ N: número.
 - ◆ T: tipo.
 - ◆ V: valor.
 - ◆ S, U, V etc: para segundos, terceros y cuartos tipos.

Generics

- ◆ En los *generics* se puede usar un parámetro para un tipo con ? Cuando es un tipo desconocido.
- ◆ Son usados para reducir las restricciones de un tipo de modo que un método pueda funcionar con una lista de *List<Integer>*, *List<Double>* y *List<Number>*.
- ◆ El término *List<Number>* es más restrictivo que *List<? extends Number>* porque el primero solo acepta una lista de *Number* y el segundo una lista de *Number* o de sus subtipos.
- ◆ *List<? extends Number>* es un *upper bounded wildcard*.
- ◆ `public static void process(List<? extends Number> list) { /* ... */ }`

Genéricos upper bounded

```
◆ public class Limite<T> {  
    ◆ private T t;  
    ◆ public void set(T t) {  
        ◆     this.t = t;  
    ◆ }  
    ◆ public T get() {  
        ◆     return t;  
    ◆ }  
    ◆ public <U extends Number> void revision(U u){  
        ◆     sout("T: " + t.getClass().getName());  
    ◆ }  
}
```

```
public static void main(String[] args) {  
    Limite<Integer> obj = new  
    Limite<Integer>();  
    obj.set(new Integer(10));  
    obj.revision("Cadena ejemplo"); // error!!  
}
```

Generics Lower Bounded

- ◆ Otro caso:
- ◆ queremos definir un método que inserte objetos *Integer* en un *List*.
- ◆ Para mayor flexibilidad queremos que ese método pueda trabajar con cualquier tipo de lista que permita contener *Integer*, ya sea *List<Integer>*, *List<Number>* y *List<Object>* Es decir, clases padre de *Integer*.
- ◆ Lo podemos conseguir definiendo *List<? super Integer>* que se conoce como *Lower Bounded Wildcard*.

PRÁCTICA: crear variante del ejercicio anterior para usar la presente estrategia

Colecciones

En Java se dispone de interfaces para trabajar con colecciones las cuales son: Collection, Set, List, Queue y Map:

- **Collection<E>**: Un grupo de elementos individuales, frecuentemente con alguna regla aplicada a ellos.
- **List<E>**: Elementos en una secuencia particular que mantienen un orden y permite duplicados. La lista puede ser recorrida en ambas direcciones con un ListIterator.

Hay 3 tipos de constructores:

- **ArrayList<E>**: Su ventaja es que el acceso a un elemento en particular es ínfimo. Su desventaja es que para eliminar un elemento, se ha de mover toda la lista para eliminar ese “hueco”.
- **Vector<E>**: Es igual que ArrayList, pero sincronizado. Es decir, que si usamos varios hilos, no tendremos de qué preocuparnos hasta cierto punto.
- **LinkedList<E>**: En esta, los elementos están conectados con el anterior y el posterior. La ventaja es que es fácil mover/eliminar elementos de la lista, simplemente moviendo/eliminando sus referencias hacia otros elementos. La desventaja es que para usar el elemento N de la lista, debemos realizar N movimientos a través de la lista.

Colecciones

- **Set<E>**: No puede haber duplicados. Cada elemento debe ser único, por lo que si existe uno duplicado, no se agrega. Por regla general, cuando se redefine equals(), se debe redefinir hashCode(). Es necesario redefinir hashCode() cuando la clase definida será colocada en un HashSet.
- **Map<K,V>**: Un grupo de pares objeto clave-valor, que no permite duplicados en sus claves. Es quizás el más sencillo, y no utiliza la interfaz Collection. Los principales métodos son: put(), get(), remove().
- **HashMap<K,V>**: Se basa en una tabla hash, pero no es sincronizado.
- **HashTable<K,V>**: Es sincronizado, aunque que no permite null como clave.
- **LinkedHashMap<K,V>**: Extiende de HashMap y utiliza una lista doblemente enlazada para recorrerla en el orden en que se añadieron. Es ligeramente más rápida a la hora de acceder a los elementos que su superclase, pero más lenta a la hora de añadirlos.
- **TreeMap<K,V>**: Se basa en una implementación de árboles en el que ordena los valores según las claves. Es la clase más lenta.

Creación e inicialización de una ArrayList

Un ArrayList es una colección de objetos redimensionable en la que están disponibles los métodos más habituales para el manejo de matrices.

Esta clase pertenece a la biblioteca java.util y contiene métodos para agregar, borrar, clorar, modificar, obtener el índice entre otros.

Ejemplo:

```
ArrayList<String> nombreArrayList = new ArrayList<String>(); // Declaración de un ArrayList  
nombreArrayList.add("Elemento"); // Añade el elemento al ArrayList  
nombreArrayList.add(n, "Elemento 2"); // Añade el elemento al ArrayList en la posición 'n'  
nombreArrayList.size(); // Devuelve el numero de elementos del ArrayList  
nombreArrayList.get(2); // Devuelve el elemento que esta en la posición '2' del ArrayList
```

Acceso a un valor de una matriz o ArrayList

Para acceder a los valores contenidos en un ArrayList existen métodos útiles para tal objetivo:

- **get(i)**: Obtiene el elemento en la posición i del ArrayList.
- **size()**: Retorna el tamaño del ArrayList.
- **contains(X)**: Retorna true si existe el elemento X en el ArrayList.

Ejemplo:

```
ArrayList <Integer> miarray= new ArrayList<>();
    for (int i = 0; i < miarray.size(); i++) {
        int value = miarray.get(i);
        System.out.println("miarray: " + value);
    }
    int count = miarray.size();
    System.out.println("Count: " + count);
```

Depuración de Errores



Depuración de Errores

- ◆ Existen muchos casos en los que se hace difícil entender lo que pasa en un programa. La solución es depurar (debug) el código del programa.
- ◆ La máquina virtual de Java, tiene capacidades de depuración de código incluso desde una máquina remota. Eclipse hace que la depuración sea sencilla.
- ◆ Depurar un programa es recorrerlo paso a paso, viendo el valor de las variables del programa. Esto nos permite ver exactamente qué está pasando en nuestro programa cuando se está ejecutando una línea de código específica.

Depuración de Errores

- ◆ El depurador permite controlar la ejecución, es decir, decidir exactamente dónde desea pausar todos los subprocessos del proceso e inspeccionar el estado en dicho punto.
- ◆ Puede dividir todo en cualquier momento, saltar las instrucciones, entrar o saltar las funciones, ejecutar hasta el cursor, editar y continuar, y la favorita de todo el mundo, establecer puntos de interrupción.

Depuración de Errores

- ◆ Un punto de interrupción es un comando que se ejecuta al llegar la ejecución de un programa a la línea especificada, y que interrumpe su ejecución en este punto.
- ◆ Los puntos de interrupción permiten investigar el comportamiento del programa en un segmento determinado: ver los valores de las variables, la pila de funciones, etcétera. Luego podemos reanudar o finalizar el proceso de depuración.
- ◆ Se debe establecer al menos un punto de interrupción en el código del programa antes de iniciar la depuración.

Depuración de Errores

- ◆ En cuanto se pause la los depuradores ofrecerán muchas maneras de inspeccionar el valor de las variables para formar o comprobar una hipótesis.
- ◆ Podemos supervisar un valor mientras revisamos el código. Esto nos permite ver las variables locales y evaluar expresiones complejas (todo sin abandonar el depurador).
- ◆ Incluso puede consultarse de manera interactiva niveles profundos de la estructura de datos.

I/O & Threads



La plataforma Java es compatible con interacción a desde la consola de comandos de dos maneras:

- **Flujos estándares**

Los flujos estándar son una característica de muchos sistemas operativos de forma predeterminada leen la entrada desde el teclado y escriben la salida a la pantalla. La plataforma Java es compatible con tres flujos estándar:

- **Entrada estándar:** Esto se utiliza para alimentar con datos al programa del usuario y por lo general un teclado se utiliza como flujo de entrada estándar y representa como **System.in**.

La plataforma Java es compatible con interacción a desde la consola de comandos de dos maneras:

- **Flujos estándares**
- **Salida estándar:** Esto se utiliza para enviar los datos producidos por el programa del usuario y por lo general una pantalla de ordenador se utiliza para flujo de salida estándar y representa como `System.out`.
- **Error estándar:** Esto se utiliza para enviar los datos de error producidos por el programa del usuario y por lo general una pantalla de ordenador se utiliza para flujo de error estándar y representa como `System.err`.

Lectura del flujo de entrada de la consola

```
import java.io.*;  
  
public class ReadConsole {  
    public static void main(String args[]) throws IOException{  
        InputStreamReader cin = null;  
        try {  
            cin = new InputStreamReader(System.in);  
            System.out.println("Ingrese texto!, 'q' para salir.");  
            char c;  
            do {  
                c = (char) cin.read();  
                System.out.print(c);  
            } while(c != 'q');  
        }finally {  
            if (cin != null) {  
                cin.close();  
            }  
        }  
    }  
}
```

Programación de tareas del sistema operativo

Java soporta la programación concurrente o multihilo, con el apoyo de concurrencia básica en el lenguaje de programación Java y las bibliotecas de clases de Java.

Desde la versión 5.0, la plataforma Java ha incluido en su API librerías de concurrencia de alto nivel en los paquetes `java.util.concurrent`.

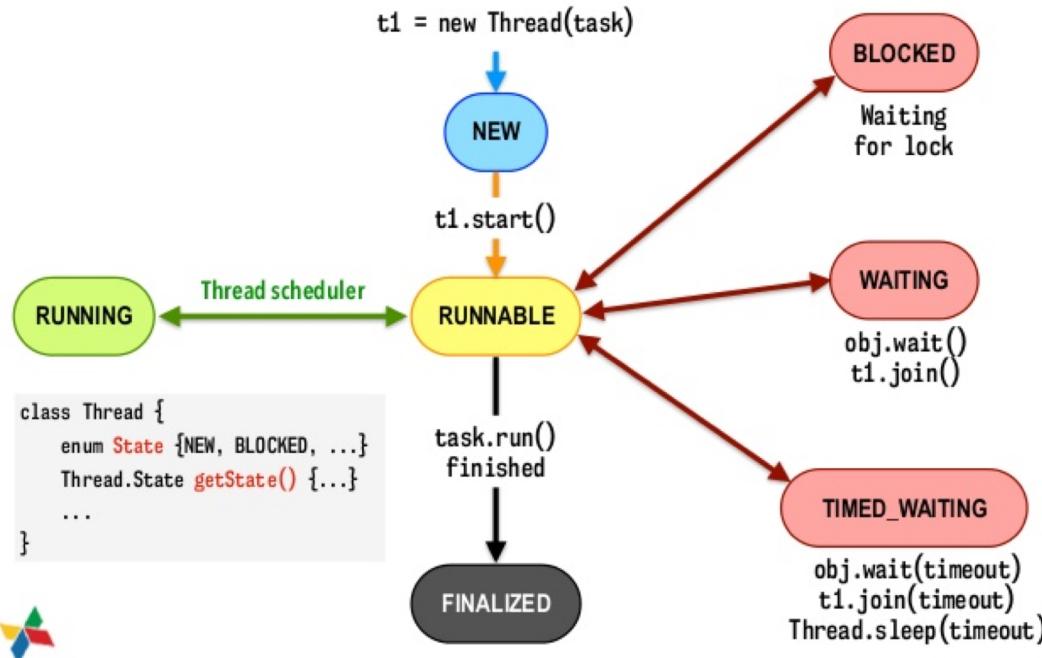
Un programa multihilo contiene dos o más partes que se pueden ejecutar al mismo tiempo y cada parte puede manejar diferentes tareas al mismo tiempo, haciendo un uso óptimo de los recursos disponibles, especialmente cuando el equipo tiene varias CPU.

Reconocimiento de entornos multithread

En la programación concurrente, hay dos unidades básicas de ejecución: procesos y subprocesos(hilos).

- **Procesos**
 - Un proceso tiene un entorno de ejecución autónomo.
 - Un proceso en general, tiene un juego completo, privado de los recursos básicos de tiempo de ejecución; en particular, cada proceso tiene su propio espacio de memoria.
-
- **Hilos**
 - La creación de un nuevo hilo requiere menos recursos que la creación de un nuevo proceso.
 - Un hilo debe existir dentro de un proceso - cada proceso tiene al menos uno.
 - Los hilos comparten los recursos del proceso, incluyendo la memoria y los archivos abiertos.
Esto hace que la comunicación sea mas eficiente, pero potencialmente problemática.

Ciclo de Vida de un hilo



Creación de soluciones multithread

Existen dos maneras de implementar multithread y crear hilos:

- Heredando de la clase Thread, y sobrescribir su método run().

```
class Hilo extends Thread{  
    public void run(){  
        ....  
    }  
}
```

- Implementando la interfaz Runnable proporcionando la implementación solo para su método run().

```
class Hilo implements Runnable{  
    public void run(){  
        .....  
    }  
}
```

Arranque de un Thread

Las aplicaciones ejecutan *main()* tras arrancar. Esta es la razón de que *main()* sea el lugar natural para crear y arrancar otros threads.

```
Hilo hijo = new Hilo("Hijo");
hijo.start();
System.out.println("main en ejecucion!");
```

Manipulación de un Thread

Todo lo que queramos que haga el thread ha de estar dentro de *run()*, por ejemplo, intentamos esperar durante una cantidad de tiempo en segundos

```
hijo.sleep(1000);
```

Suspensión de un Thread

Puede resultar útil suspender la ejecución de un thread sin marcar un límite de tiempo. Para este tipo de control de thread se puede utilizar el método suspend().

```
hijo.suspend();
```

Este método no detiene la ejecución permanentemente. El thread es suspendido indefinidamente y para volver a activarlo de nuevo necesitamos realizar una invocación al método resume

```
hijo.resume();
```

Parada de un Thread

El último elemento de control que se necesita sobre threads es el método *stop()*. Se utiliza para terminar la ejecución de un thread:

```
hijo.stop();
```

Stop() no destruye el thread, sino que detiene su ejecución. La ejecución no se puede reanudar con *start()*. Cuando se liberen las variables que se usan en el thread, el objeto thread quedará marcado para eliminarlo y el *garbage collector* se encargará de liberar la memoria que utilizaba.

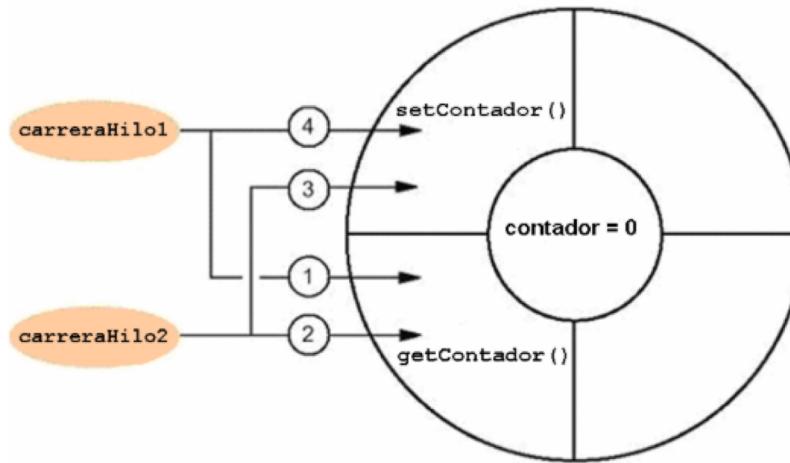
Si se necesita, se puede comprobar si un thread está vivo o no; considerando vivo un thread que ha comenzado y no ha sido detenido.

```
hijo.isAlive();
```

Uso compartido de datos en threads

A la hora de acceder a datos comunes los thread necesitan cierto orden para asegurarse que los hilos concurrentes no interfieren entre si y operan correctamente con los datos.

En la figura de ejemplo se observa 2 hilos que acceden al mismo recurso y como no esta sincronizado el acceso a ese dato compartido presenta inconsistencia.



Después de ejecutar ambos hilos y actualizar contador, esta incorrectamente en 1 en vez de 2

Sincronización e interbloqueos

- Los hilos se comunican principalmente mediante el intercambio de acceso a los campos y los objetos de referencia se refieren a los campos.
- Esta forma de comunicación es muy eficiente, pero hace dos tipos de errores posibles:
 - Interferencia hilo
 - Errores de coherencia de memoria.
- La herramienta necesaria para evitar estos errores es la sincronización.
- Sin embargo, la sincronización puede introducir discordia de hilo, que se produce cuando dos o más hilos intentan acceder al mismo recurso al mismo tiempo y pueden hacer que el tiempo de ejecución de Java para ejecutar uno o más hilos sea más lento, o incluso suspender su ejecución.

Sincronización

La clave para la **sincronización en Java** es el concepto de **monitor**, que controla el acceso a un objeto. Un *monitor* funciona implementando el concepto de **bloqueo** (*lock*). Cuando un objeto está bloqueado por un hilo, ningún otro hilo puede obtener acceso al objeto. Cuando el hilo sale, el objeto está desbloqueado y está disponible para ser utilizado por otro hilo.

Todos los objetos en Java tienen un monitor. Esta característica está integrada en el lenguaje Java en sí. Por lo tanto, **todos los objetos se pueden sincronizar.** La sincronización está respaldada por la palabra clave **synchronized** y algunos métodos bien definidos que tienen todos los objetos.

Sincronización

Podemos sincronizar el acceso a un método modificándolo con la palabra clave **synchronized**. Cuando se llama a ese método, el hilo de llamada entra en el monitor del objeto, que luego bloquea el objeto.

- Mientras está bloqueado, ningún otro hilo puede ingresar al método, o ingresar cualquier otro método sincronizado definido por la clase del objeto.
- Cuando el hilo retorna del método, el monitor desbloquea el objeto, permitiendo que sea utilizado por el siguiente hilo. Por lo tanto, la sincronización se logra con prácticamente ningún esfuerzo de programación de tu parte.

```
class sumArray{  
    private int sum;  
  
    //sumArray está sincronizado  
    synchronized int sumArray(int nums[]){  
        sum=0;  
        for (int i=0; i<nums.length;i++){  
            sum+=nums[i];  
            System.out.println("Total acumulado de "+Thread.currentThread().getName()+" es  
"+sum);  
        try {  
            Thread.sleep(10);//permitir el cambio de tarea  
        }catch (InterruptedException exc){  
            System.out.println("Hilo interrumpido");  
        }  
    }  
    return sum;  
}
```

```
class MiHilo implements Runnable{  
    Thread hilo;  
    static sumArray sumarray= new sumArray();  
    int a[];  
    int resp;  
  
    //Construye un nuevo hilo.  
    MiHilo(String nombre, int nums[]){  
        hilo= new Thread(this,nombre);  
        a=nums;  
    }  
  
    //Un método que crea e inicia un hilo  
    public static MiHilo creaEInicia (String nombre,int nums[]){  
        MiHilo miHilo=new MiHilo(nombre,nums);  
  
        miHilo.hilo.start(); //Inicia el hilo  
        return miHilo;  
    }  
    //Punto de entrada del hilo  
    public void run(){  
        int sum;  
        System.out.println(hilo.getName()+" iniciando.");  
  
        resp=sumarray.sumArray(a);  
        System.out.println("Suma para "+hilo.getName()+" es "+resp);  
        System.out.println(hilo.getName()+" terminado.");  
    }  
}
```



```
class Sincronizacion {  
    public static void main(String[] args) {  
        int a[]={1,2,3,4,5};  
        MiHilo mh1 = MiHilo.creaEInicia("#1",a);  
        MiHilo mh2 = MiHilo.creaEInicia("#2",a);  
  
        try {  
            mh1.hilo.join();  
            mh2.hilo.join();  
        }catch (InterruptedException exc){  
            System.out.println("Hilo principal interrumpido.");  
        }  
    }  
}
```





Gracias!

Alguna pregunta?

Pueden contactarnos mediante
info@impactotecnologico.net y apuntarse a
algunos de nuestros cursos



