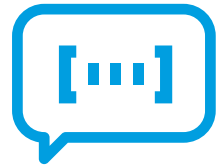
netmind
a bts company

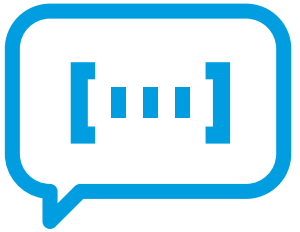
Java 11



1. La plataforma JAVA
2. Herramientas de desarrollo
3. Sintaxis básica
4. Strings y Arrays
5. Flujos de control
6. Clases y objetos
7. Composición y herencia
8. Empaquetamiento
9. Interfaces
10. Excepciones
11. Colecciones
12. Aserciones, Anotaciones, Expresiones Lambda
13. Streams de E/S

netmind
a bts company

01



LA PLATAFORMA JAVA

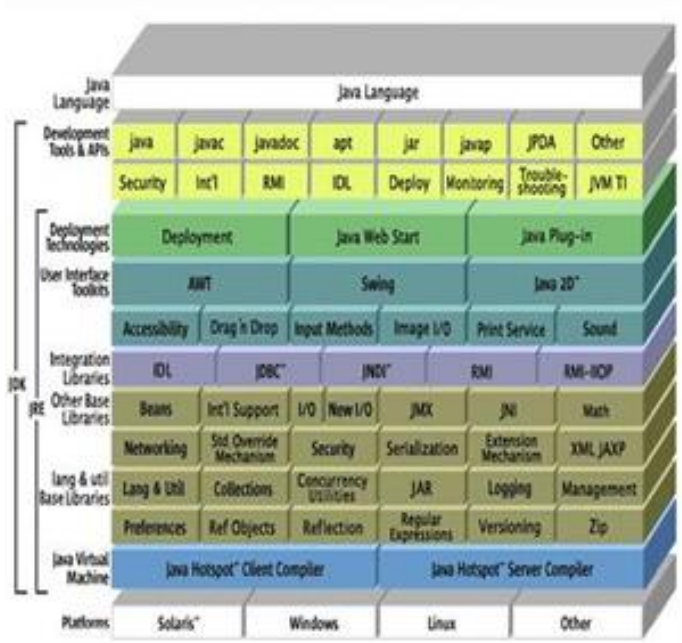
Java

- Java es un lenguaje de programación y una plataforma informática comercializada por primera vez en 1995 por Sun Microsystems.
- La Tecnología Java incluye:
- Un lenguaje de programación
- Un entorno de desarrollo: Compilador (javac), Intérprete (java), Generador de documentación (javadoc), herramienta para empaquetar los .class, …
- Es una aplicación: programas de uso general que funcionan en cualquier máquina donde el Java runtime enviroment (JRE)
- Un entorno de ejecución: JRE

Java

LENGUAJE DE PROGRAMACIÓN + PLATAFORMA

```
19      public static void hasilBagi(double x, double y) {
20          DecimalFormat angka = new DecimalFormat("###,###");
21          double c = x/y;
22          System.out.println
23              ("Hasil Dari "+angka.format(x)+" : "+
24                  angka.format(y)+" = "+angka.format(c));
25      }
26      public static void hasilKurang(double x, double y) {
27          DecimalFormat angka = new DecimalFormat("###,###");
28          double c = x-y;
29          System.out.println
30              ("Hasil Dari "+angka.format(x)+" - "+
31                  angka.format(y)+" = "+angka.format(c));
32      }
33
```

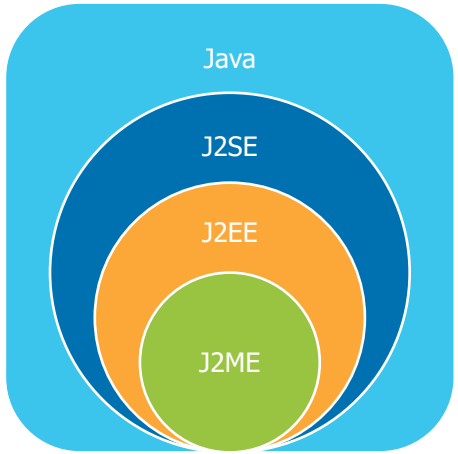


ORIENTADO A OBJETOS

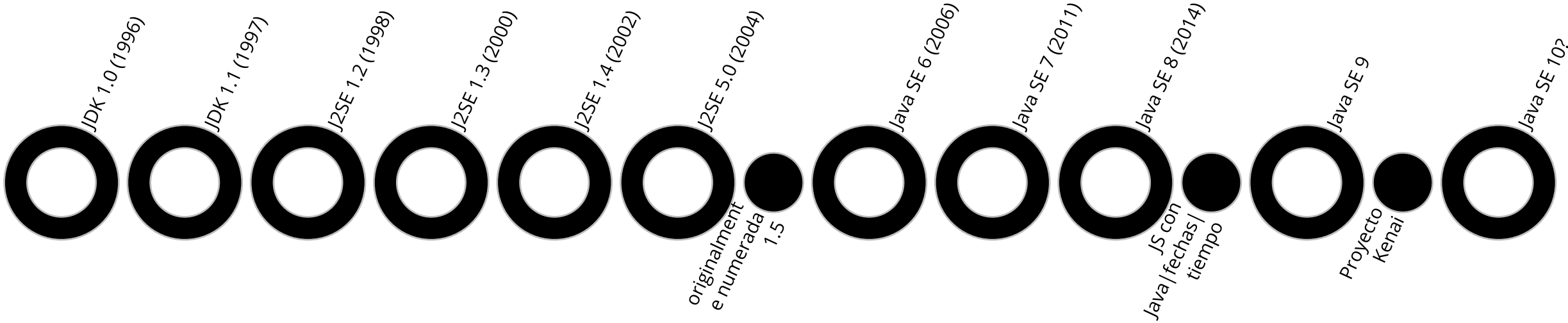
RÁPIDO, SEGURO Y FIABLES

Distribuciones

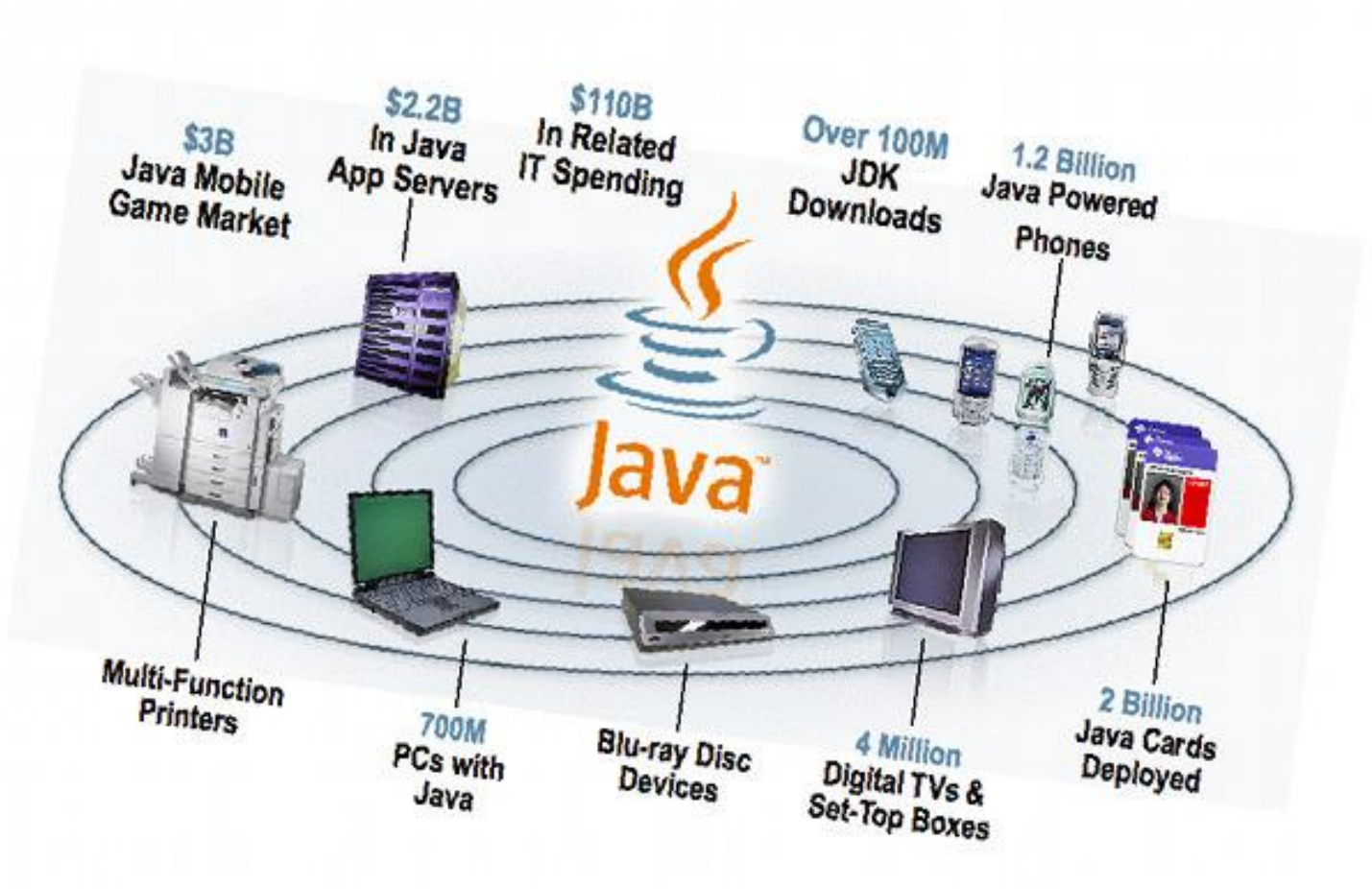
- J2SE o simplemente Java SE: Java 2 Standard Edition o Java Standard Edition
- J2EE o JEE: Java 2 Enterprise Edition
- J2ME: Java 2 Micro Edition



Evolución


JDK 1.0 (1996)
JDK 1.1 (1997)
J2SE 1.2 (1998)
J2SE 1.3 (2000)
J2SE 1.4 (2002)
J2SE 5.0 (2004)
originalment
e numerada
1.5
Java SE 6 (2006)
Java SE 7 (2011)
Java SE 8 (2014)
JS con
Java|fechas|
tiempo
Java SE 9
Proyecto
Kenai
Java SE 10?

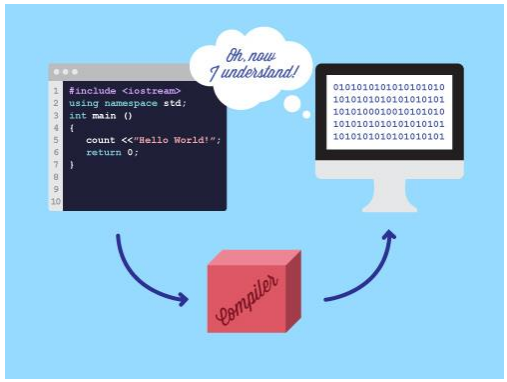
En qué se usa?



JAVA ESTÁ POR TODAS PARTES!

Ciclo de vida de un programa JAVA

Filename.**java**
(source file)



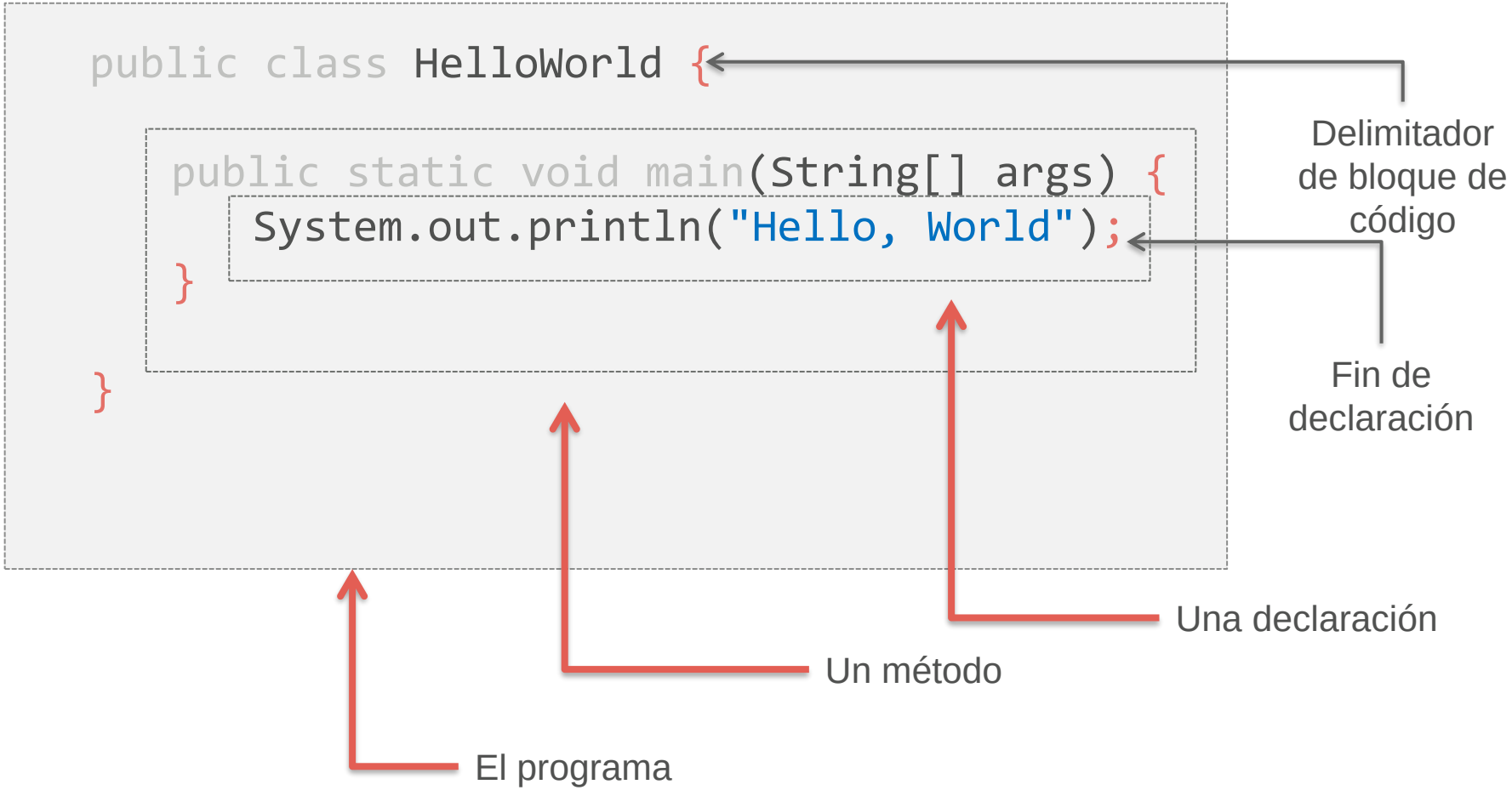
2. COMPILA

Filename.**class**
(Bytecode file)

1. ESCRIBE EL CÓDIGO

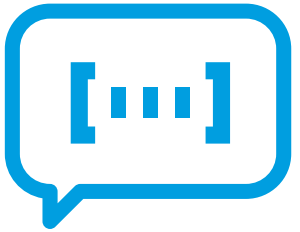
3. EJECUTA EN EL ENTORNO JRE

Un programa Java básico



netmind
a bts company

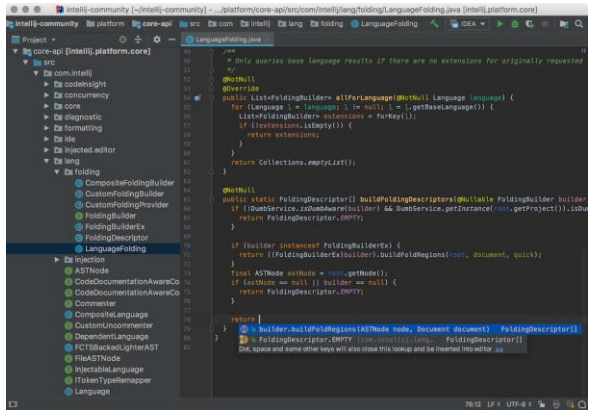
02

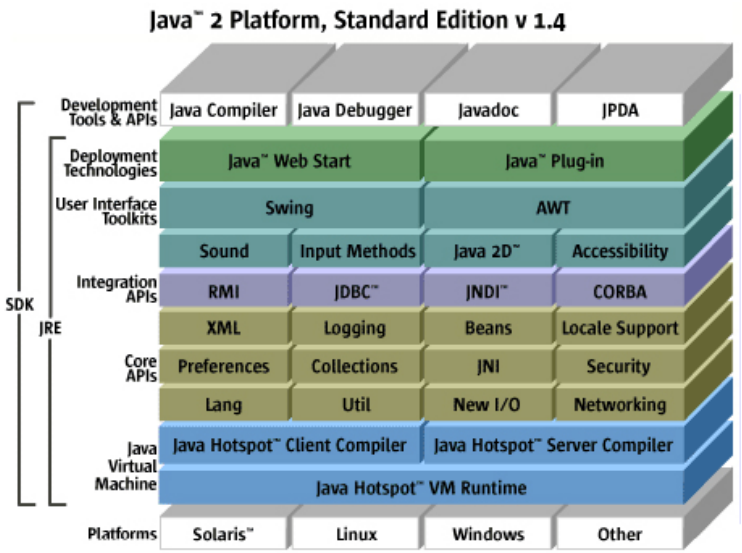


HERRAMIENTAS DE DESARROLLO

Herramientas de desarrollo necesarias

Editor de texto o IDE + Java Development Kit + Java API





eclipse

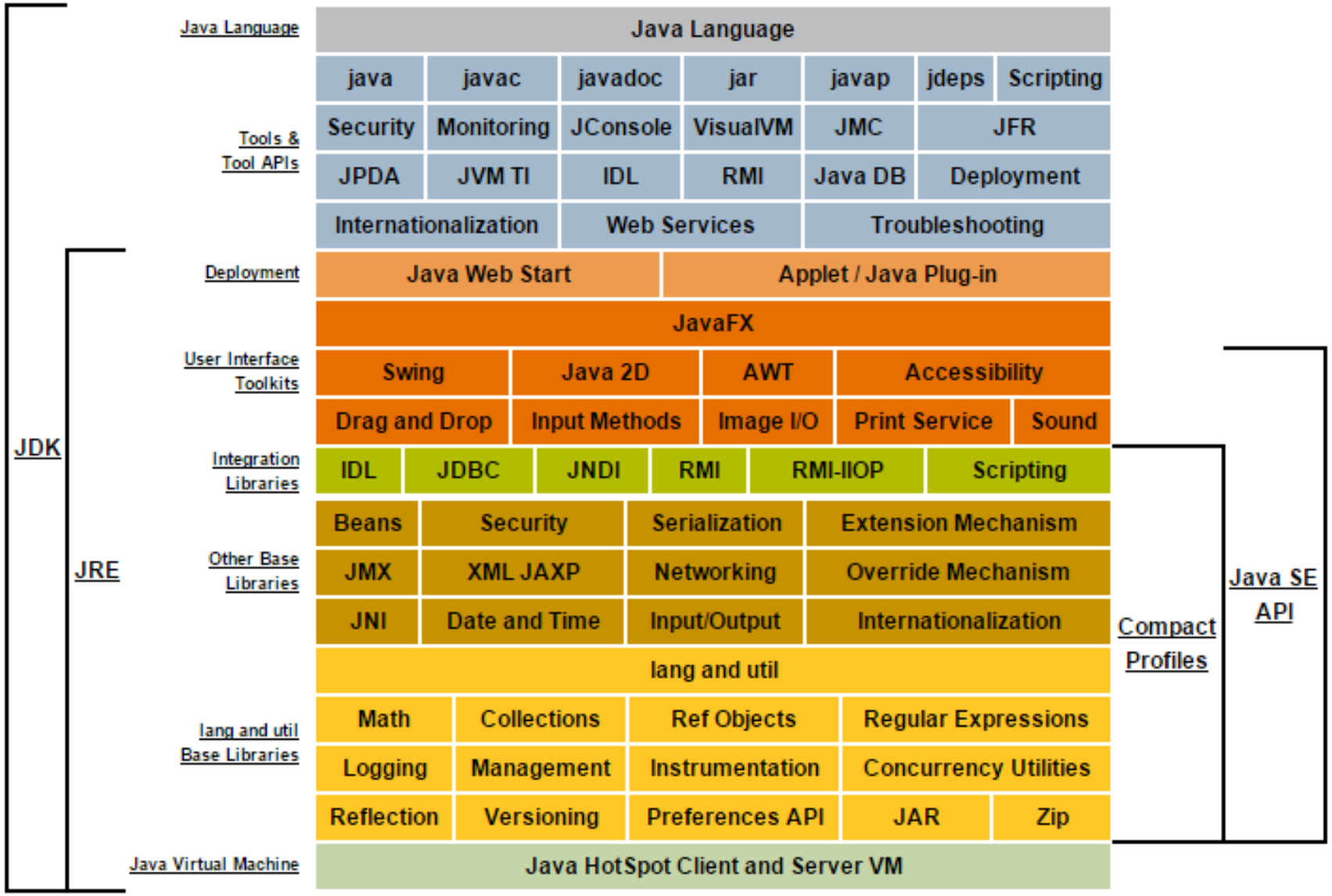
IJ

API Java

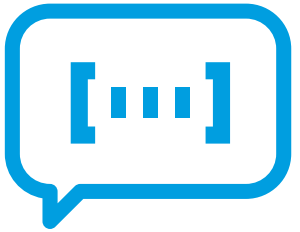
- Al instalar Java (el paquete JDK), además del compilador y la máquina virtual de Java se instalan bastantes más elementos.
- Entre ellos, una cantidad muy importante de clases que están a disposición de los programadores, listas para ser usadas.
- Estas clases junto a otros elementos forman lo que se denomina API (Application Programming Interface) de Java.
- La documentación de dicha API la podemos encotrar en:
- https://docs.oracle.com/en/java/javase/11/

API Java

Description of Java Conceptual Diagram



03



SINTAXIS BÁSICA JAVA

Identificadores

- Un identificador es el nombre que podemos asignar a determinados elementos en el código fuente del programa, como por ejemplo a las variables, funciones, etc.
- Para crear un identificador debemos tener en cuenta las siguientes reglas:
  - Debe comenzar por una letra, aunque el resto del nombre puede contener también números.
  - Se pueden usar tanto mayúsculas como minúsculas, pero ten en cuenta que dependiendo del lenguaje de programación usado, se hace distinción entre ambas: 'Comision', 'comision' y 'COMISION'.
  - No se pueden usar tildes.
  - También es posible usar el carácter de subrayado '_'.
- Algunos identificadores válidos: 'x' , 'el_nombre', 'salario', 'mes1'...
- Algunos identificadores no válidos: '3po' (comienza con un número), '3+4' (contiene una carácter no válido), 'comisión' (no se admiten tildes).

Palabras clave

- Los lenguajes de programación usan internamente algunas palabras, denominadas palabras clave o palabras reservadas que no podemos usar como identificadores.

- Por ejemplo, algunas palabras clave o reservadas del lenguaje Java son:

```
abstract.
assert.
boolean.
break.
byte.
case.
catch.
char
....
```

Variables y Constantes

- En cualquier lenguaje de programación necesitaremos usar variables y constantes para almacenar determinados valores y así poderlos recuperarlos posteriormente para usarlos en otra parte del código fuente.
  - **Constantes**: se declaran al principio del programa escribiendo su nombre en mayúsculas. Se les puede asignar su valor una sola vez
  - **Variables**: se declaran al principio del programa. Su valor puede ser modificado en cualquier punto del código fuente del programa.
- Para usar una variable o constante es preciso declararla (crearla) antes de poder inicializarla (darle un valor)
  - Se pueden hacer ambas cosas en la misma línea del código fuente.
- En java para declarar una variable o constante es necesario indicar su tipo

Variables y Constantes

- En java para declarar una variable o constante es necesario indicar su tipo

```
int foo;
double d1, d2;
boolean isFun;
```

```
int foo= 42;
double d1 = 3.14, d2 = 2 * 3.14;
boolean isFun = true;
```

```
static final int SIMPLE = 0, ONE_SPRING = 1, TWO_SPRING = 2;
```

Tipos primitivos

Type	Definition
boolean	true or false
char	16-bit, Unicode character
byte	8-bit, signed, two's complement integer
short	16-bit, signed, two's complement integer
int	32-bit, signed, two's complement integer
long	64-bit, signed, two's complement integer
double	64-bit, 1.7e-308 – 1.7e+308, floating point value
float	32-bit, 3.4e-38 -- 3.4e+38, floating-point value

Expresiones

Asignación

```
int x=7;
int x=2, y;
y=x;
```

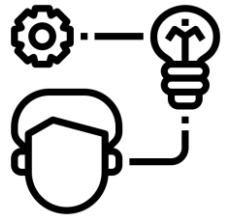
Evaluación

```
x = 3+4;
```

Operadores

- Operadores: Asignación
  - x+=y
  - x%=y
  - x *= y
  - x /= y
- Comparación
  - Igual (==)
  - No igual (!=)
  - Mayor (>)
  - Mayor o igual (>=)
  - Menor (<).
  - Menor o igual (<=)
- Aritméticos
  - Módulo (%)
  - Incremento (++)
  - Decremento (- -)
  - Negativo (-)
- Lógicos
  - and *&&*, or //, not *!*
- Operadores especiales
  - Condicional: **?** (expresión ternaria)
    - (evaluación)? <caso verdadero>:<caso false>;
    - a = isTrue? x : y ;

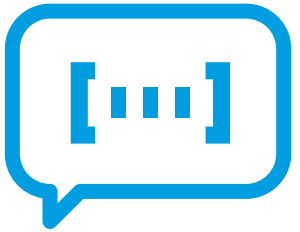
Pongámoslo en práctica



- Crea un programa que sume dos números y muestre el resultado por consola (usa 3 variables)
- Prueba con enteros, flotantes y doble flotante

netmind
a bts company

04



STRINGS Y ARRAYS

Strings

- Los Strings son clases Java! → por tanto tienen propiedades y métodos
  - greeting.length(); //the string length
  - greeting.concat(String); //concatenate with another string
    - Para concatenar Strings también se puede usar: string1+string2
  - greeting.replace(char,char); //replace 1st char by 2nd char
- Para darle un valor inicial lo indicaremos mediante comillas
  - "<valor> "
- http://docs.oracle.com/javase/11/docs/api/java/lang/String.html

```
String greeting = "Hello world!";
```

Arrays

- Objeto de contenedor que contiene un número fijo de valores de un solo tipo.
- Se definen usando corchetes después del tipo:
  - *<tipo>[] <nombre>*
- Si queremos darles un valor, podemos hacerlo usando las llaves
  - <tipo>[] <nombre> = {valor_1, valor_2,…};
- También podemos optar por indicar su tamaño:
  - <tipo>[] <nombre> = new <tipo> [<tamaño>];

```
String[] months = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
```

```
int[] myIntArray = new int[3];
int[] myIntArray = {1,2,3};
int[] myIntArray = new int[]{1,2,3};
```

Arrays

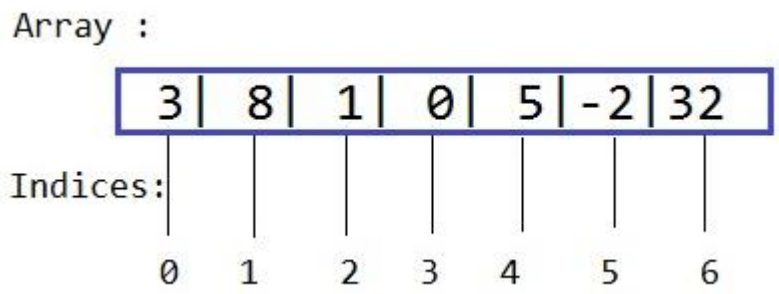
- Los arrays también son objetos, por tanto tiene propiedades y métodos
  - int[] intArray = { 1, 2, 3, 4, 5 };
  - int tam = intArray .length;
  - String intArrayString = Arrays.toString(intArray);

  - String[] stringArray = { "a", "b", "c", "d", "e" };
  - boolean b = Arrays.asList(stringArray).contains("a");

- https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

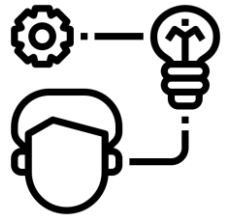
Acceso a los datos de un Array

- Para acceder a los elementos de un array usaremos el índice del elemento dentro del array
- Tanto para obtener su valor como para asignárselo




```
int[] anArray = new int[10];

anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
…
System.out.println("Element at index 0: “ + anArray[0]);
System.out.println("Element at index 2: " + anArray[1]);
```

Pongámoslo en práctica

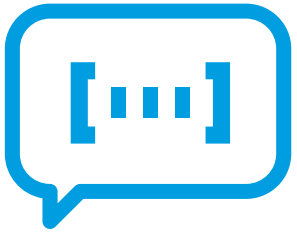


- Crea un programa que contenga dos variables:
  - Una con los meses del año
  - Otra con el nombre de los días de la semana

- Muestra por consola (en este orden) los meses correspondientes a los meses enero, noviembre, julio, diciembre, marzo; y los días lunes, viernes, martes.

- Se deben mostrar separados por un espacio

05



FLUJOS DE CONTROL

Condicionales if/else

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```

```
void applyBrakes() {

   // Si se está moviendo,  bajar la velocidad;
   // si no es que ha parado
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already
stopped!");
    }

}
```

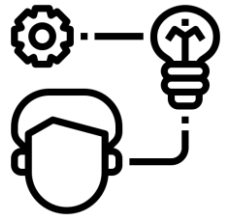
```
int testscore = 76;
char grade;

if (testscore >= 90) {
    grade = 'A';
} else if (testscore >= 80) {
    grade = 'B';
} else if (testscore >= 70) {
    grade = 'C';
} else if (testscore >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}
System.out.println("Grade = " + grade);
```

Condicionales switch

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......

default:
 code to be executed if
all cases are not matched;
}
```

```
int month = 8;
String monthString;
switch (month) {
    case 1:  monthString = "January";
             break;
    case 2:  monthString = "February";
             break;
    case 3:  monthString = "March";
             break;
    case 4:  monthString = "April";
             break;
    case 5:  monthString = "May";
             break;
    case 6:  monthString = "June";
             break;
    case 7:  monthString = "July";
             break;
    case 8:  monthString = "August";
             break;
    case 9:  monthString = "September";
             break;
    case 10: monthString = "October";
             break;
    case 11: monthString = "November";
             break;
    case 12: monthString = "December";
             break;
    default: monthString = "Invalid month";
             break;
}
System.out.println(monthString);
```

Pongámoslo en práctica

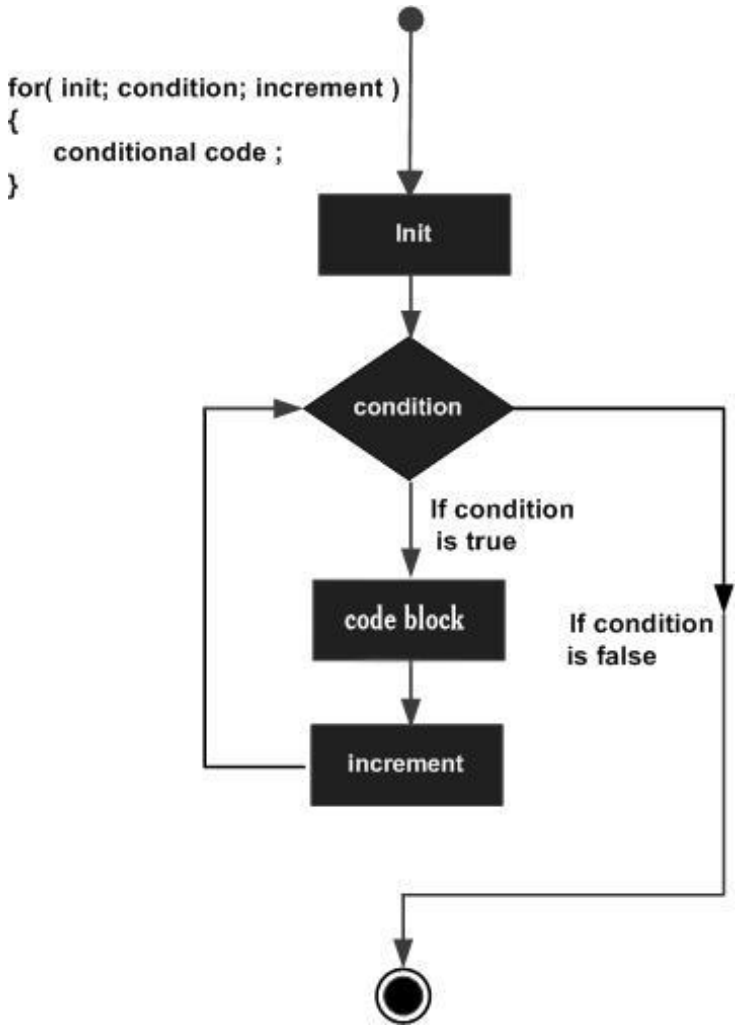


- Crea un programa que contenga una variable String con un valor cualquiera.
- Evalúa si la variable vale “Hola” e indica por consola si tiene el valor o no (“La variable vale Hola” o ”La variable no vale hola”).
- Tip: usa el método .equals()

For Loop

- Permite repetir las sentencias en su interior mientras se cumpla la condición; e incrementando un contador

```
for(initialization;condition;incr/decr){
//code to be executed
}
```



For Loop

```
public class ForExample {


public static void main(String[] args) {

      for(int i=1;i<=10;i++){
        System.out.println(i);
      }
   }

}
```

```
int[] list ={1, 2, 3, 4, 5, 6, 7, 8,
9, 10};
int total = 0;

for(int i = 0; i < list.length; i++){
    total += list[i];
    System.out.println( list[i] );
}

System.out.println(total);
```

For-each Loop

- Una variación para recorrer listas o arrays es el loop for-each

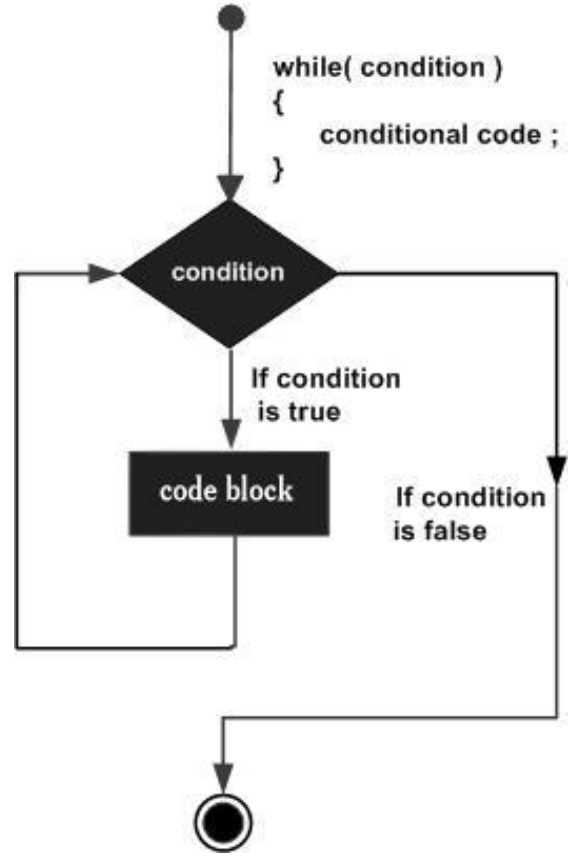
```
for(declaration : expression) {
//Statements
}
```

```
int arr[]={12,23,44,56,78};
for(int i:arr){
        System.out.println(i);
}
```

While Loop

- Se usa para iterar una parte del programa varias veces. Si el número de iteración no es fijo, se recomienda utilizar while loop.

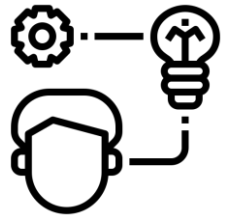
```
while(condition){
//code to be executed
}
```



While Loop

```
public class Test {

   public static void main(String args[]) {
      int x = 10;

      while( x < 20 ) {
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }
   }
}
```

Pongámoslo en práctica



- Cree un programa denominado LetterCounter que:
- Contenga internamente un array de Strings
- Muestre por consola cada valor del array y su longitud.
  - Un valor y longitud por linea

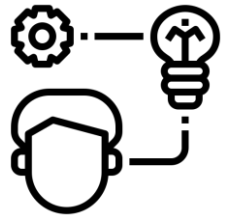
- Tips:
- La longitud de un texto lo da el método length()

Argumentos en main

- En la ejecución de un programa se pueden pasar argumentos.
- Estos vienen en forma de array de String en la variable **args**

```
public class Echo {

    public static void main (String[] args) {
        for (String unArgumento: args) {
            System.out.println(unArgumento);
        }
    }

}
```

Pongámoslo en práctica



- Crea un programa denominado WordCounter que:
- Contenga internamente un array de Strings
- Reciba como argumento un valor numérico
- Muestre por consola solo los valores del array que tengan la misma longitud de letras que el primer argumento pasado al programa.

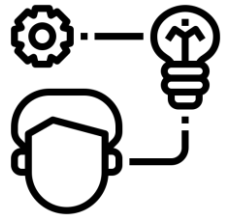
- **Tips**: Los argumentos se reciben como String, es necesario convertirlos a int:
  - int result = Integer.parseInt(“234”);

Try – catch – finally

- El bloque **try** contiene sentencias dentro del cual puede producirse una excepción.
  - Las **excepciones** son errores en tiempo de ejecución (cuando se esta ejecutando el programa)
- Un bloque try siempre es seguido por un bloque catch, que maneja la excepción que se produce en el bloque try asociado.
- Un bloque **try-catch** puede ser seguido por un bloque finally, que se ejecuta siempre en cualquiera de las circunstancias.

```
try {
    float aResult=aVar1/aVar2;
} catch ( Exception ex) {
    System.out.println("An Exception:“+ex);
} finally {
    System.out.println( "finally") ;
}
```

Pongámoslo en práctica



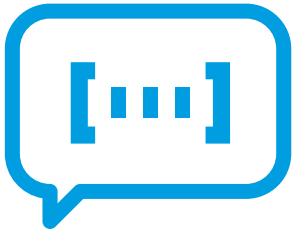
- Modifica WordCounter para que pase dos parámetros:
  - El primero: el número de caracteres.
  - El segundo: un texto.

- El programa debe mostrar sólo las palabras del array con la longitud del primer parámetro y sólo si esa palabra contiene el segundo parámetro.
- ¡Recuerda tener cuidado con las excepciones!

- Tips: para saber si un String contiene otro: aString.contains (anotherString);

netmind
a bts company

06



CLASES Y OBJETOS

Clases y objetos en Java

- Una clase de objetos describe las características comunes a un conjunto de objetos.
- Durante la ejecución de la aplicación se producirá la instanciación de esta clase, es decir, la creación de los objetos que representan cada uno de los individuos con sus características propias, es decir, valores específicos para sus atributos

Clase

```
class <class_name>{
    field;
    method;
}
```

Objeto

```
<class_name> instane = new <class_name>();
```

Clases y objetos en Java

```
class Student{
 int id;
 String name;
}
```

```
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);
 }
}
```

Clases y objetos en Java

- En Java la implementación de las operaciones se realiza en el interior de la definición de la clase, justo tras su declaración

```
class Student{
    int rollno;
    String name;

    void insertRecord(int r, String n){
      rollno=r;
      name=n;
    }

    void displayInformation(){
        System.out.println(rollno+" "+name);
    }

}
```

public, private, protected

- El principio de ocultación de información se plasma en los lenguajes OO en diversos mecanismos de protección de los miembros de la clase
- Podemos etiquetar tanto clases, como propiedades como métodos de clase con un nivel de protección

- **+public:** Visible para todo el mundo
- **-private:** Visible solo para los elementos internos de la clase
- **#protected:** Visible para el paquete y subclases

```
public class Car{
    public speed;
    private motorbrand;
    protected tankbrand;
    ….
    public int getSpeed(){…}
}
```

Constructores

- La iniciación de los atributos de la clase se realiza en Java mediante el uso de constructores cuyo nombre coincide con el de la clase

```
<class_name>(){}
```

```
class Student{
    int id;
    String name;

    Student(int i,String n){
        id = i;
        name = n;
    }
    ….
}
```

```
Student s1 = new
Student(111,"Karan");
Student s2 = new
Student(222,"Aryan");
```

Sobrecarga de Constructores

- Java permite la sobrecarga de operaciones, por tanto se pueden definir varios constructores posible para una clase siempre que se diferencien en la lista de argumentos

```
class Student{
    int id;
    String name;
    int age;

    Student(int i,String n){
        id = i;
        name = n;
    }

    Student(int i,String n,int a){
        id = i;
        name = n;
        age=a;
    }

}
```

```
Student s1 = new
Student(111,"Karan");
Student s2 = new
Student(222,"Aryan",25);
```

this

this es una variable de referencia que se refiere al objeto actual.

- this se puede usar para referir la variable de instancia de clase actual.
- this se puede usar para invocar el método de clase actual (implícitamente)
- this() se puede usar para invocar al constructor de la clase actual.
- this se puede pasar como un argumento en la llamada al método.
- this se puede pasar como argumento en la llamada del constructor.
- this se puede usar para devolver la instancia de clase actual del método.

```
class Student{
     int rollno;
     String name;
     float fee;

     Student(int rollno,String name,float fee){
          this.rollno=rollno;
          this.name=name;
          this.fee=fee;
     }
     ...
}
```

static keyword

- Los atributos o métodos estáticos (static) son aquellos que pertenecen a la clase en sí, y no a los objetos
- De un atributo estático no se genera una copia por cada objeto que se crea.
- **Existe una única copia compartida por todos los objetos de la clase**
- Una operación estática únicamente puede acceder a miembros estáticos

final keyword

- La palabra clave final se usa para restringir el usuario
- Si haces una variable final, no se puede cambiar su valor: Será constante.

```
final int speedlimit=90; //final variable
```

- Si haces un método final, no se puede sobreescribir.

```
class Bike{
  final void run(){System.out.println("running");}
}
```

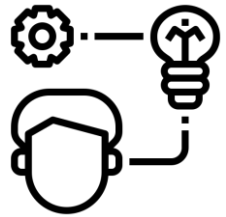
- Si haces una clase final, no se podrá extender. **final class** Bike{}

```
final class Bike{}
```

static keyword

```
class Student{
    int rollno;
    String name;
    static String college ="ITS";

    Student(int r,String n){
        rollno = r;
        name = n;
    }

     public final static void display (){
        System.out.println(rollno+" "+name+" "+college);
    }

}
```

Pongámoslo en práctica

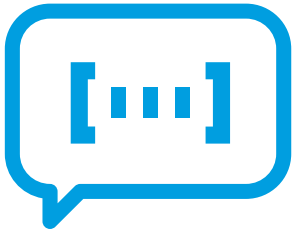


- Modela una clase que represente a un coche en Java
  - Atributos: marca, tipoCoche y velocidad
  - Métodos: tiempo=avanzar(n_kilometros):double, parar(),…pueden haber más métodos
  - Constructor: debe permitir inicializar todos los atributos
- Crea un programa que genere diversos coches y determine cuál de ellos es el más rápido en función del tiempo que tarda en recorrer una distancia de 100 kilómetros.

- Tip: para dividir dos enteros y generar un double:
  - int distancia, velocidad;
  - double tiempo=(double) distancia/(double)velocidad;

[···] netmind
a bts company

07



COMPOSICIÓN Y HERENCIA

Relaciones entre clases

- A nivel de diseño, podemos distinguir entre 5 tipos de relaciones básicas entre clases de objetos: dependencia, asociación, agregación, composición y herencia*
- Las primeras cuatro se expresan como atributos de clase o variables dentro de métodos

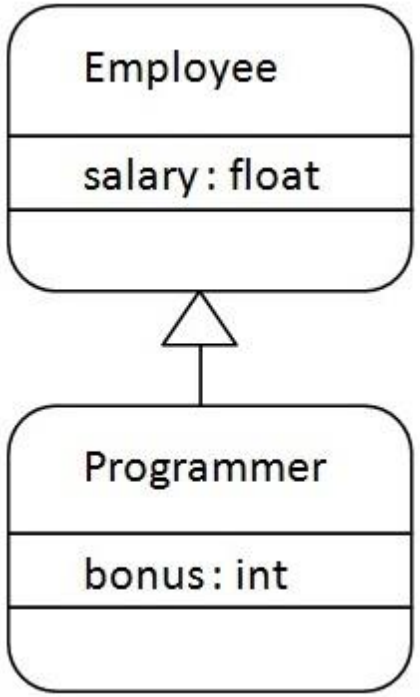
```
class Employee{
    int id;
    String name;
    Address address;//Address is a class
    ...

    void fillAddress(String street, int number,…){
        this.address = new Address(street, number,…);
        …
    }
}
```

Herencia

- La herencia en java es un mecanismo en el cual un objeto adquiere todas las propiedades y comportamientos del objeto padre.
- La idea detrás de la herencia en java es que se puede crear nuevas clases que se basan en las clases existentes.
- Cuando se hereda de una clase existente, puede volver a utilizar métodos y campos de la clase padre y también puede agregar nuevos métodos y campos.

```
class Subclass-name extends Superclass-name {
   //methods and fields
}
```

Herencia




```
class Employee{
float salary=40000;
}

class Programmer extends Employee{
int bonus=10000;
}
```

```
public class Client{
public static void main(String args[]){
     Programmer p=new Programmer();
     System.out.println("Programmer salary
is:"+p.salary);
     System.out.println("Bonus of Programmer
is:"+p.bonus);
}
}
```

Herencia

```
class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}

class TestInheritance{
    public static void main(String args[]){
        Dog d=new Dog();
        d.bark();
        d.eat();
    }
}
```

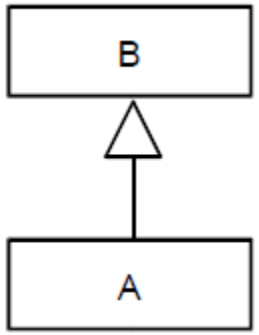
Super

- La palabra clave super es una variable de referencia que se usa para referirse al objeto clase padre inmediato.
- Siempre que se crea la instancia de subclase, se crea una instancia de clase primaria implícitamente a la que se hace referencia por super.

```
class Animal{
String color="white";
}

class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color); //prints color of Dog class
System.out.println(super.color); //prints color of Animal
class
}
}
```

Polimorfismo

- El polimorfismo (o upcasting) en Java es un concepto mediante el cual podemos realizar una sola acción de diferentes maneras ("poly":muchos, "morphs":formas).
- El polimorfismo consiste en la posibilidad de que una referencia a objetos de una clase pueda conectarse también con objetos de descendientes de ésta



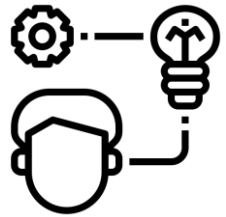

```
B rb = new B (); // Asignación ordinaria

A ra = rb; // Asignación polimorfa
```

Polimorfismo

```
class Bank{
     float getRateOfInterest(){return 0;}
}

class SBI extends Bank{
     float getRateOfInterest(){return 8.4f;}
}

class ICICI extends Bank{
     float getRateOfInterest(){return 7.3f;}
}

class AXIS extends Bank{
     float getRateOfInterest(){return 9.7f;}
}
```

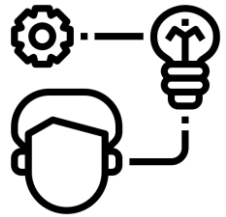
```
class TestPolymorphism{
     public static void main(String args[]){
          Bank b;
          b=new SBI();
          System.out.println("SBI Rate of Interest:
          "+b.getRateOfInterest());
          b=new ICICI();
          System.out.println("ICICI Rate of Interest:
          "+b.getRateOfInterest());
          b=new AXIS();
          System.out.println("AXIS Rate of Interest:
          "+b.getRateOfInterest());
     }
}
```

Pongámoslo en práctica



- Modifica el programa de coches y modela distintos tipos de coches (deportivos, familiares, todoterrenos)
  - Cada tipo de coche nuevo heredará a la clase coche
  - Cada tipo de coche tendrá atributos propios especiales, como por ejemplo: los todoterrenos tendrán 1 o 2 cambios
  - Asimismo, pueden tener distintos métodos o implementaciones distintas del mismo método (por ejemplo de avanzar o parar)
- Crea un programa que genere diversos coches de distintos tipos y determine cuál de ellos es el más rápido en recorrer una distancia de 100 kilómetros

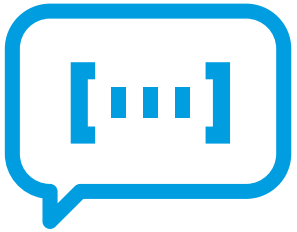
Pongámoslo en práctica



- Modela el motor con una clase nueva
  - Atributos: consumo, número de pistones y cilindrada
  - Métodos: arrancar(), mover() y parar()
- Modifica la clase Coche (del programa anterior) para incorporar esta clase como un atributo y relaciónalo en el método de avanzar.

netmind
a bts company

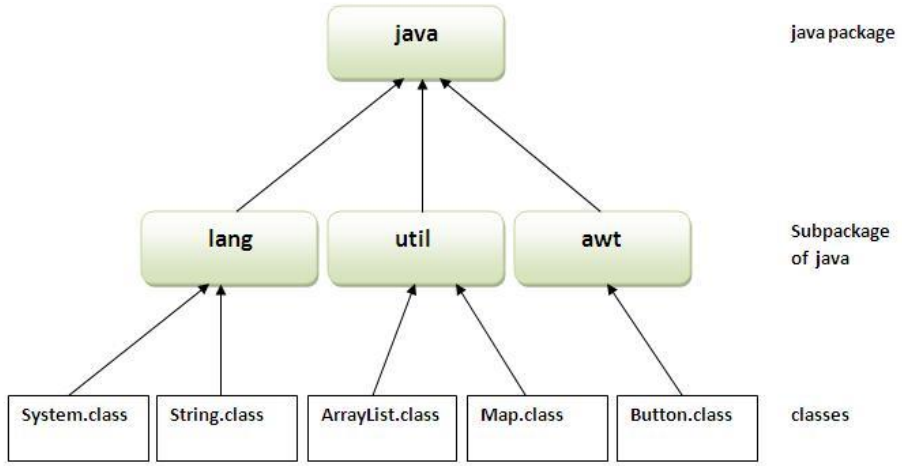
08



EMPAQUETAMIENTO

Empaquetamiento

- Un paquete es un grupo de tipos similares de clases, interfaces y subpaquetes.
- El paquete en java se puede categorizar de dos maneras, paquete incorporado (de fábrica) y paquete definido por el usuario.
- Hay muchos paquetes incorporados tales como java, lang, awt, javax, oscilación, red, io, util, sql etc.



Empaquetamiento

- Ventajas del empaquetado en Java
- Un paquete de Java se utiliza para categorizar las clases y las interfaces de modo que puedan ser mantenidas fácilmente.
- Un paquete Java proporciona protección de acceso.
- Un paquete Java elimina la colisión de nombres.

Nomenclatura de paquetes

Nombre de dominio	Prefijo del nombre de paquete
hyphenated-name.example.org	org.example.hyphenated_name
example.int	int_.example
123name.example.com	com.example._123name

```
package package_path; //declarando el paquete en cada clase
Import package_path.*; //importando todas las clases de un paquete
Import package_path.Concrete_Class; //importando una clase concreta
```

Estructura de directorio de paquetes

- Por cada nivel de un paquete se debe crear un directorio

```
int_.example → int/example/
com.example._123name → _123name/example/com
```

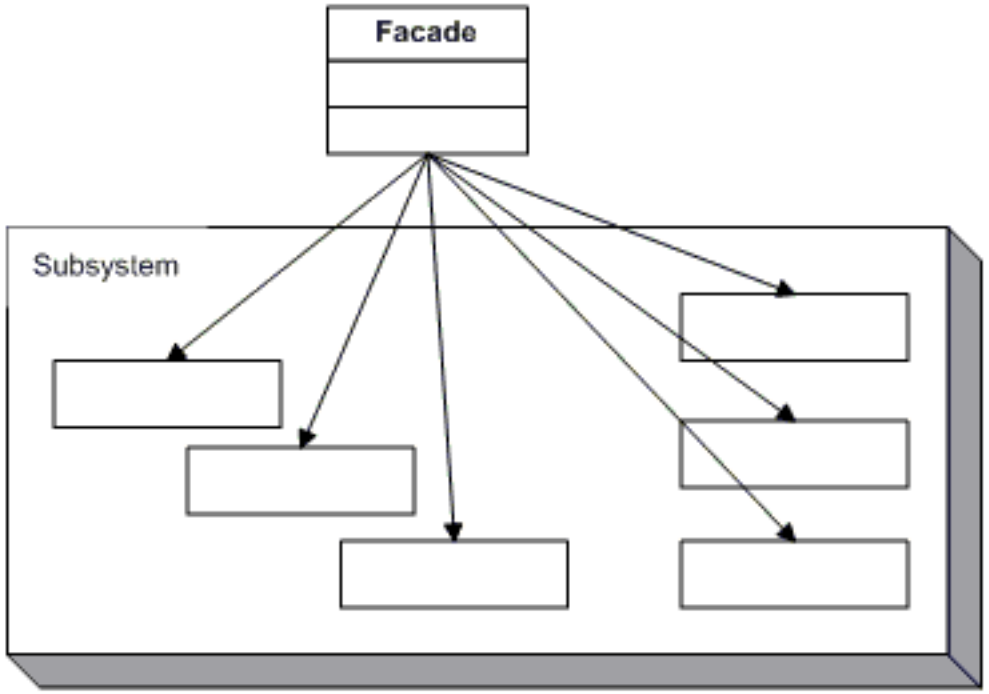
- Dentro de dichos directories se crearán las clases

```
int_.example.MyClass → int/example/MyClass.java
```

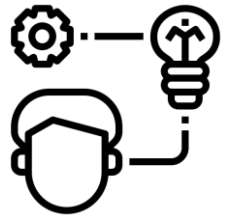
- Para que un clase cliente (que usa clases de un paquete) funcione es necesario compilar primero las clases del paquete

Aplicaciones orientadas a objetos

- En una aplicación orientada a objetos debe existir una clase que represente la propia aplicación. Este sería el punto donde comenzaría la ejecución de la misma
- Este esquema de diseño responde a un patrón llamado Façade

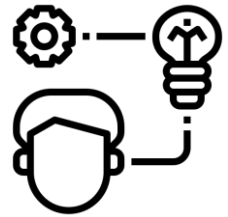


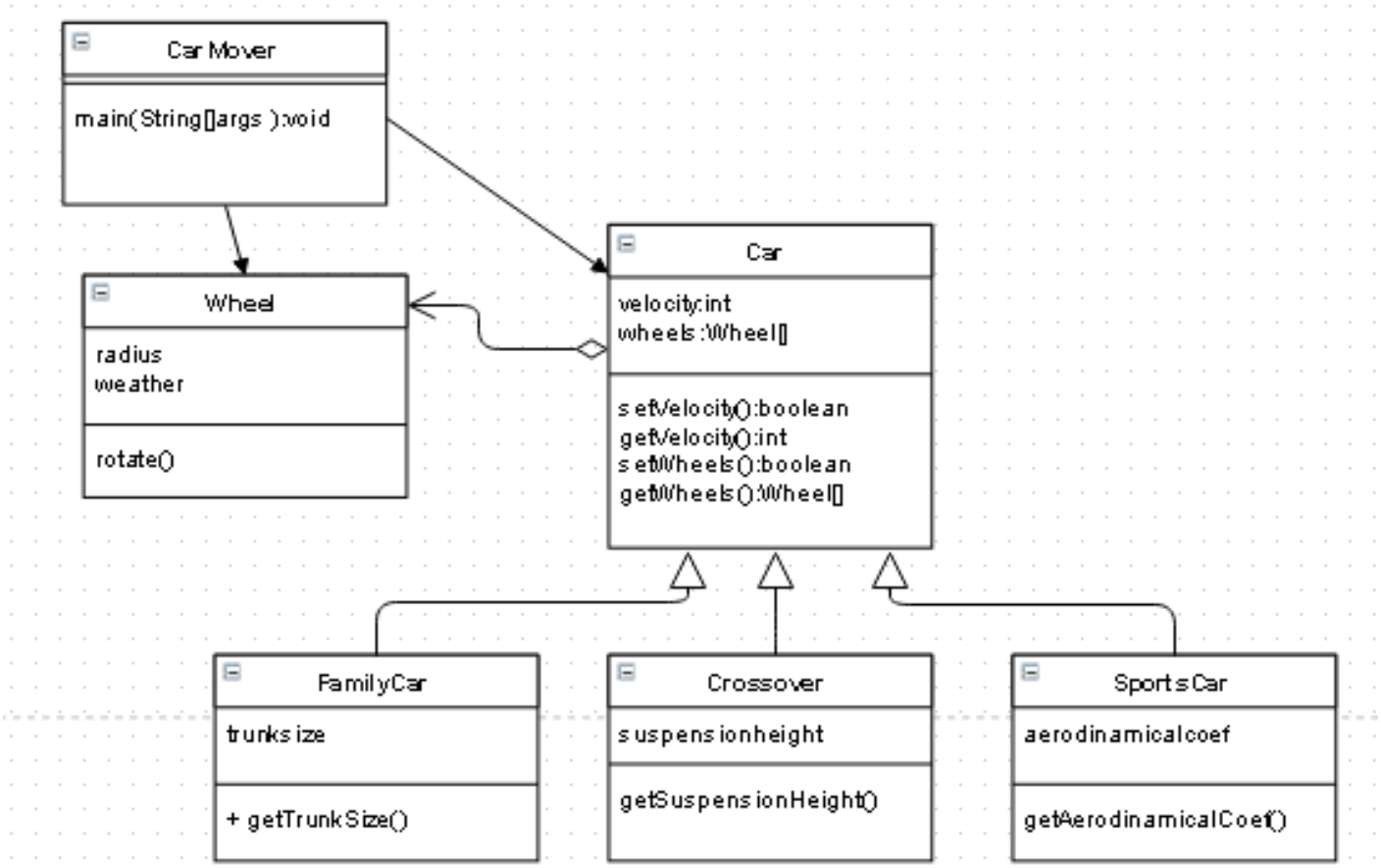
Pongámoslo en práctica



- Crea una aplicación respondiendo al diseño que se presenta en la siguiente diapositiva (continuación del programa Coche).
- Genera los paquetes necesarios y posiciona las clases dentro de dichos paquetes (indicado en los cuadros por colores).
- Recuerda generar el paquete base con el dominio de tu marca

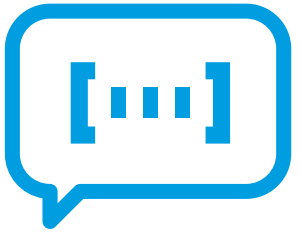
Pongámoslo en práctica





netmind
a bts company

09



INTERFACES

Abstracción en Java

- La abstracción es el proceso de ocultar los detalles de la implementación y mostrar sólo la funcionalidad al usuario.
- Muestra sólo las cosas importantes para el usuario y oculta los detalles internos,
- por ejemplo, para enviar SMS's, sólo tienes que escribir el texto y enviar el mensaje. No conoces el procesamiento interno sobre la entrega de mensajes.
- Abstracción le permite centrarse en lo que hace el objeto en lugar de cómo lo hace.

- En Java se pueden lograr la abstracción usando
  - Interfaces
  - Clases abstractas

Interfaces

- Una interfaz es una plantilla de una clase. Es un mecanismo para lograr la abstracción.
- Tiene constantes estáticas y métodos abstractos.
- Las interfaces serán implementadas por clases concretas
- Porqué usar interfaces:
  - Para lograr abstracción.
  - Mediante interface podemos implementar herencia múltiple.
  - Para limitar el acoplamiento (loose coupling).

```
Interface A{
    method_name();
    method_name();
    method_name();
    …
}
```

```
Class B implements A {
    method_name(){
        …
    }
}
```

Interfaces

```
interface printable{
    void print();
}
```

Definimos la interface

Declaramos los métodos

```
class A6 implements printable{

    public void print(){
        System.out.println("Hello");
    }

}
```

La clase A6 implementa la interface

Al implementar la interface, es obligatorio que implemente los métodos declarados en la interface

Clases abstractas

- Una clase declarada como abstract se le conoce como clase abstracta
- Puede tener métodos abstractos y no abstrasctos

```
abstract class A{
    abstract void method_name();

    void non_abstract_method(){
        ….
    }

}
```

```
class B extends A{
    void method_name(){
        …
    }

}
```

Clases abstractas

```
abstract class Bike{
  abstract void run();
}
```

Definimos la clase abstracta

Declaramos los métodos abstractos

```
class Honda4 extends Bike{
    void run(){
        System.out.println("running safely..");
    }

}
```

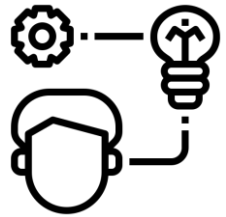
La clase Honda4 extiende la clase abstracta

Al extender la clase abstracta, es obligatorio que implemente los métodos abstractos

Clases abstractas e Interfaces

- Clases abstractas: campos que no son estáticos y finales, y definen métodos públicos, protegidos y privados.
- Interfaces: todos los campos son automáticamente públicos, estáticos y finales, y todos los métodos que declara o define (como métodos predeterminados) son públicos.
- Puede extender sólo una clase, sea o no abstracta.
- Puede implementar cualquier número de interfaces.

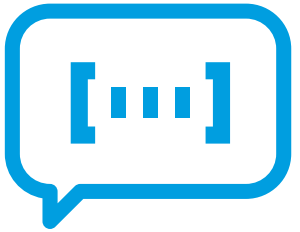
Pongámoslo en práctica



- Queremos desacoplar la Clase Coche de la clase Motor
  - Genera los mecanismos necesarios para que desde la clase Coche se use una interface de motor
- Ahora queremos que la clase Coche determine que el método avanzar() debe ser implementado por sus herederos, pero dejando que estos implementen el cuerpo del mismo, ya que depende de su cada tipo
  - Usa una clase abstracta para coche

netmind
a bts company

10

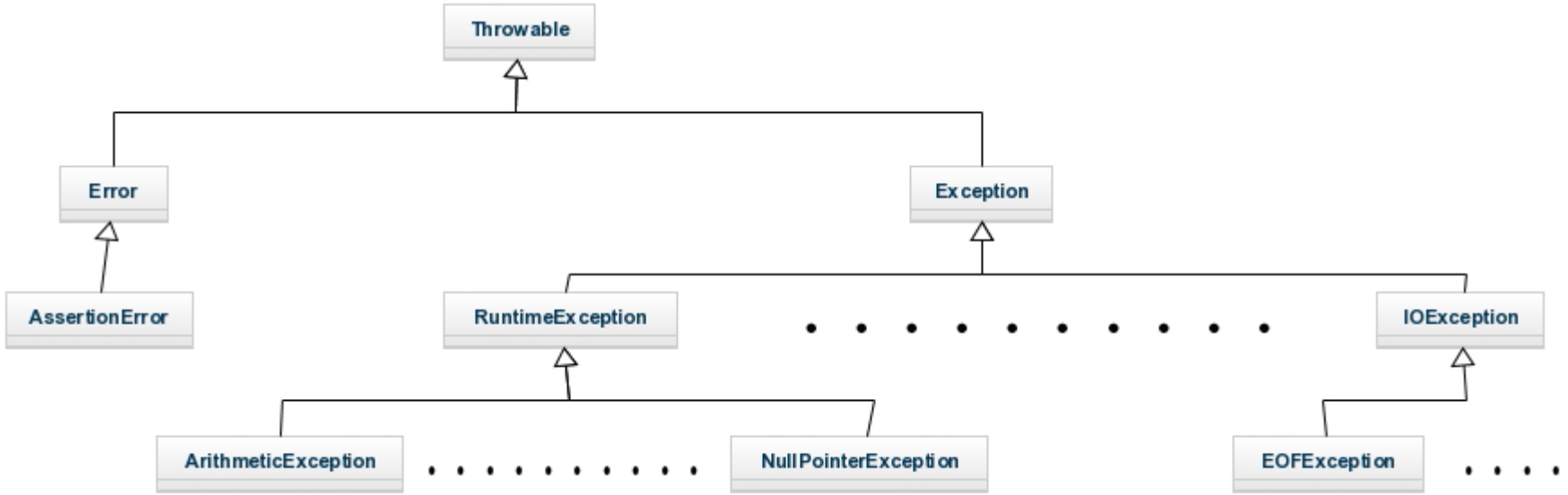


EXCEPCIONES

Excepciones en Java

- En Java los errores en tiempo de ejecución se denominan **excepciones**, y esto ocurre cuando se produce un error en alguna de las instrucciones de nuestro programa, como por ejemplo cuando se hace una división entre cero, cuando un objeto es 'null' y no puede serlo, cuando no se abre correctamente un fichero, etc.
- Cuando se produce una excepción se muestra en la pantalla un mensaje de error y finaliza la ejecución del programa.
- Cuando en Java se produce una excepción se crear un objeto de una determina clase (dependiendo del tipo de error que se haya producido), que mantendrá la información sobre el error producido y nos proporcionará los métodos necesarios para obtener dicha información.
  - Estas clases tienen como clase padre la clase **Throwable.**

Excepciones en Java



Excepciones en Java

- Java nos permite hacer un control de las excepciones para que nuestro programa no se pare inesperadamente y aunque se produzca una excepción, nuestro programa siga su ejecución, usandola estructura "try – catch – finally".

```
try {
    // Instrucciones cuando no hay una excepción
} catch ( Exception ex) {
    // Instrucciones cuando se produce una excepcion
} finally {
    // Instruciones que se ejecutan, tanto si hay como sino hay
excepciones
}
```

Declaración de excepciones personalizadas

- Para nuestros propios programas, a veces es útil que creemos nuestra excepción a medida. Para ello, tenemos que declarar una nueva clase que herede de Exception. Por ejemplo:

```
public class BadPostCodeException extends Exception {

    public BadPostCodeException() {
        super();
    }
    public BadPostCodeException(String message) {
        super(message);
    }
}
```

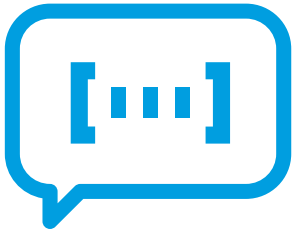
Lanzar una excepción

- Cuando en un método necesitamos lanzar una excepción, utilizaremos la palabra clave throw, seguida de una instancia de la excepción a lanzar.

```
if (postCode.length() != 5) {
    throw new BadPostCodeException("Postcodes must have 5 digits");
}
```

netmind
a bts company

11



COLECCIONES

Colecciones en Java

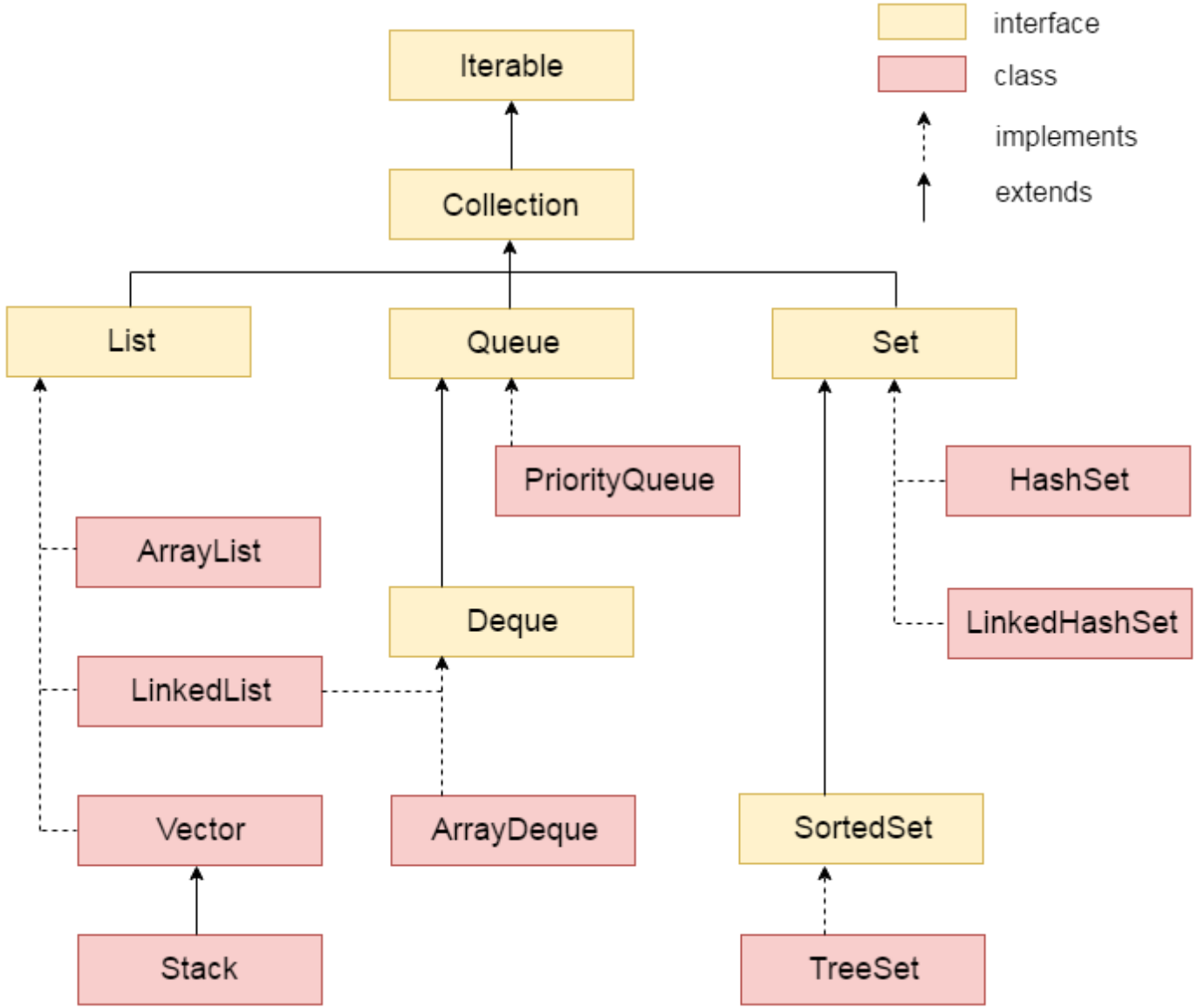
- Las colecciones es un framework que proporciona una arquitectura para almacenar y manipular el grupo de objetos.
- Son susceptibles de realizar operaciones sobre los datos que contienen, como búsqueda, clasificación, inserción, manipulación, eliminación, etc., se pueden realizar por las colecciones.
- El framework Java Collection proporciona varias interfaces (Set, List, Queue, Deque, etc.) y clases (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet, etc.).
- https://docs.oracle.com/javase/11/docs/api/java/util/Collections.html

Abstracción en Java

- La abstracción es el proceso de ocultar los detalles de la implementación y mostrar sólo la funcionalidad al usuario.
- Muestra sólo las cosas importantes para el usuario y oculta los detalles internos,
- por ejemplo, para enviar SMS's, sólo tienes que escribir el texto y enviar el mensaje. No conoces el procesamiento interno sobre la entrega de mensajes.
- Abstracción le permite centrarse en lo que hace el objeto en lugar de cómo lo hace.

- En Java se pueden lograr la abstracción usando
  - Interfaces
  - Clases abstractas

Jerarquía del framework de colecciones



Métodos comunes de colecciones

Method	Description
public boolean add(Object element)	is used to insert an element in this collection.
public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
public boolean remove(Object element)	is used to delete an element from this collection.
public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
public int size()	return the total number of elements in the collection.
public void clear()	removes the total no of element from the collection.
public boolean contains(Object element)	is used to search an element.
public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
public Iterator iterator()	returns an iterator.
public Object[] toArray()	converts collection into array.
public boolean isEmpty()	checks if collection is empty.
public boolean equals(Object element)	matches two collection.
public int hashCode()	returns the hashcode number for collection.

Iterador

- La interfaz **Iterator** proporciona un método de iterar los elementos de una colección
- La dirección de iteración siempre es de avance (hacia adelante)
- Sus métodos son:

Method	Description
public boolean hasNext()	returns true if iterator has more elements.
public object next()	returns the element and moves the cursor pointer to the next element.
public void remove()	removes the last elements returned by the iterator. It is rarely used.

Listas

- Utiliza una matriz dinámica para almacenar elementos.
- Hereda la clase AbstractList e implementa la interfaz List
- La nueva colección genérica permite tener sólo un tipo de objeto en la colección.

```
List<String> list = new ArrayList<String>();
list.add(aString);
list.set(index,aObject);
myString=list.get(index);
```

Iterador:

- Usando la interfaz Iterator
- Ejemplo usando for-each

```
for (ListIterator<Type> it =
list.listIterator(list.size()); it.hasPrevious(); ) {
    Type t = it.previous(); ...
}
```

Listas - ejemplo

```
import java.util.*;
class TestCollection1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();
//Creating arraylist
  list.add("Ravi");//Adding object in arraylist
  list.add("Vijay");
  list.add("Ravi");
  list.add("Ajay");
  //Traversing list through Iterator
  Iterator itr=list.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
import java.util.*;
class TestCollection2{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  for(String obj:al)
    System.out.println(obj);
 }
}
```

Vectores

- Vector implementan la interfaz de lista y mantienen el orden de inserción.
- Hay varias diferencias entre las clases ArrayList y Vector

ArrayList	Vector
ArrayList is not synchronized.	Vector is synchronized.
ArrayList increments 50% of current array size if number of element exceeds from its capacity.	Vector increments 100% means doubles the array size if total number of element exceeds than its capacity.
ArrayList is not a legacy class, it is introduced in JDK 1.2.	Vector is a legacy class.
ArrayList is fast because it is non-synchronized.	Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.
ArrayList uses Iterator interface to traverse the elements.	Vector uses Enumeration interface to traverse the elements. But it can use Iterator also.

Vectores

- Definición

```
Vector v = new Vector(3);
v.addElement(anObject);
Vector<String> vec = new
Vector<String>(2);
```

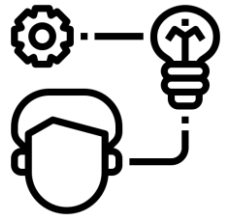
- Iteración

```
Enumeration en = vec.elements();
while(en.hasMoreElements()) {
    System.out.print(en.nextElement()
+ " ");
}
```

Vectores - ejemplo

```
import java.util.*;
class TestVector1{
 public static void main(String args[]){
  Vector<String> v=new Vector<String>();//creating vector
  v.add("umesh");//method of Collection
  v.addElement("irfan");//method of Vector
  v.addElement("kumar");
  //traversing elements using Enumeration
  Enumeration e=v.elements();
  while(e.hasMoreElements()){
   System.out.println(e.nextElement());
  }
 }
}
```

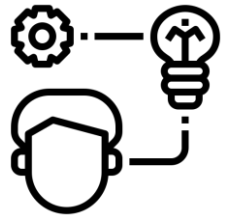
Pongámoslo en práctica



PROCESADOR DE TEXTOS

- Se quiere procesar cientos de textos y almacenar el resultado en memoria.
- El procesado del texto consistirá en transformar sus mayúsculas y minúsculas. Habrán 4 opciones:
  - Todo minúsculas.
  - Todo mayúsculas.
  - Tipo oración (la primera letra en mayúscula y el resto minúscula).
  - La primera letra de cada palabra en mayúscula.
- Los textos pueden tener cualquier longitud, pero no pueden ser vacíos.
- Los textos se poder procesar individualmente (seleccionando un elemento por sudeben índice), un subconjunto de la lista (indicando los índices de inicio y fin) o todos los textos.
- Un texto procesado se puede querer reprocesar.

Pongámoslo en práctica



PROCESADOR DE TEXTOS - TIPS:

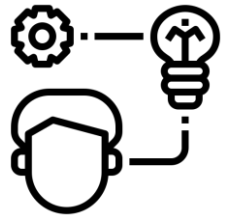
- Métodos String: toLowerCase, toUpperCase, split, substring.
  - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html
- Capitalizar la primera letra: buscar en google "Java Capitalize first letter of a String"
- Método array útil: join
  - https://stackoverflow.com/questions/1978933/a-quick-and-easy-way-to-join-array-elements-with-a-separator-the-opposite-of-sp

HashMap

- La clase **HashMap** implementa la interfaz Map utilizando una tabla hash.
- Hereda la clase AbstractMap e implementa la interfaz de Map.
- Detalles importantes:
  - Un HashMap contiene valores basados en la clave.
  - Las claves son elementos únicos.
  - Puede tener una clave null y varios valores null
  - No mantiene ningún orden.

```
import java.util.*;
class TestCollection13{
 public static void main(String args[]){
  HashMap<Integer,String> hm=new HashMap<Integer,String>();
  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");
  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

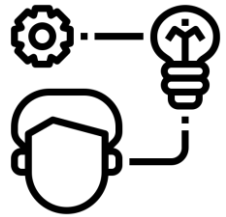
Pongámoslo en práctica



PROCESADOR DE TEXTOS

- Se quiere modificar el programa procesador de textos para que cada texto tenga una palabra clave que lo defina (como en un diccionario).
- Modifica el programa para que permita las mismas funcionalidades previas, pero con este nuevo requerimiento.

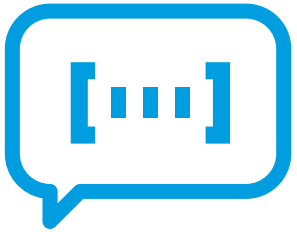
Pongámoslo en práctica



- Queremos que el programa principal use un mecanismo dinámico que le permita ir añadiendo coches a comparar.
- Cambia el programa para que esto sea posible y que pueda determinar cuál es el más rápido en recorrer los 100km

netmind
a bts company

12



ASERCIONES, ANOTACIONES, EXPRESIONES LAMBDA

Aserciones

- Una aserción es una declaración que se usa para probar suposiciones sobre el programa (por ejemplo de valor).
- Mientras se ejecuta la aserción, se considera que es verdadera. Si la aserción falla, JVM lanza un error llamado AssertionError.
- Las aserciones se usan principalmente para el propósito de pruebas.
- Para habilitar las aserciones, se debe usar el parámetro de ejecución **-ea** or **-enableassertions** (en la VM arguments)

```
assert expression;
```

```
assert expression1 : expression2;
```

Aserciones - ejemplo

```
import java.util.Scanner;

class AssertionExample{

 public static void main( String args[] ){

  Scanner scanner = new Scanner( System.in );
  System.out.print("Enter your age ");

  int value = scanner.nextInt();
  assert value>=18:" Not valid";

  System.out.println("value is "+value);
 }

}
```

Anotaciones

- Una **anotación** es una etiqueta que representa metadatos
- Se adjunta a la clase, la interfaz, los métodos o los campos para indicar alguna información adicional que puede ser utilizada por el compilador java y JVM.
- Hay varias anotaciones integradas en java. Algunas anotaciones se aplican al código java
  - @Override
  - @SuppressWarnings
  - @Deprecated
- y otras, a otras anotaciones.
  - @Target
  - @Retention
  - @Inherited
  - @Documented

```
@Annotation
class aClass{…}
```

```
class aClass{
    @Annotation
    aField;

    @Annotation
    aMethod(){…}
}
```

@Override

- La anotación @Override asegura que el método de subclase reemplazará el método de clase padre.
- Si no es así, se produce error de tiempo de compilación:
  - Por ejemplo, ante un error de ortografía, etc
  - Por lo tanto, es mejor marcar la anotación @Override que proporciona la seguridad de que el método es anulado.

```
class Animal{
     void eatSomething(){System.out.println("eating something");}
}

class Dog extends Animal{
     @Override
     void eatsomething(){System.out.println("eating foods");}//should be eatSomething
}

class TestAnnotation1{
     public static void main(String args[]){
          Animal a=new Dog();
          a.eatSomething();
     }
}
```

@SuppressWarnings

- Se utiliza para suprimir las advertencias emitidas por el compilador.

```
import java.util.*;
class TestAnnotation2{
    @SuppressWarnings("unchecked")
    public static void main(String args[]){
        ArrayList list=new ArrayList();
        list.add("sonoo");
        list.add("vimal");
        list.add("ratan");

        for(Object obj:list)
        System.out.println(obj);

    }
}
```

@Deprecated

- Indica que el método está obsoleto y por tanto puede ser eliminado en versiones futuras.

```
class A{
    void m(){System.out.println("hello m");}

    @Deprecated
    void n(){System.out.println("hello n");}
}

class TestAnnotation3{
    public static void main(String args[]){

        A a=new A();
        a.n();
    }
}
```

Enum

- Enum es un tipo de datos que contiene conjunto fijo de constantes.
- Puede usarse para los días de la semana (DOMINGO, LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES y SÁBADOS), direcciones (NORTE, SUR, ORIENTE y OESTE) etc.
- Las constantes del enum son estáticas y finales implícitamente.
- Los Enums de Java se pueden considerar como clases que tienen conjunto fijo de constantes.

```
class EnumExample1{
    public enum Season { WINTER, SPRING, SUMMER, FALL }

    public static void main(String[] args) {
        for (Season s : Season.values())  System.out.println(s);
    }
}
```

Expresiones Lambda

- La expresión Lambda es una característica nueva e importante de Java SE 8 +.
- Proporciona una forma clara y concisa de representar una interfaz de método utilizando una expresión, reemplazando a las antiguas clases internas anónimas.

```
(argument-list) -> {body}
```

- Es muy útil para las colecciones: Ayuda a iterar, filtrar y extraer datos de una colección.

Interfaz funcional

- Las expresiones Lambda implementan una interfaz funcional.
- Una interfaz que tiene sólo un método abstracto se denomina interfaz funcional.
- Java proporciona la anotación @FunctionalInterface, que se utiliza para declarar dicha interfaz funcional.

Expresiones Lambda: Sintaxis

- La manera de declara una expresión lambda es mediante la sintaxis

```
(argument-list) -> {body}
```

  - Argument-list: Puede estar vacío.
  - Arrow-token "->": Se utiliza para vincular la lista de argumentos y el cuerpo de expresión.
  - Cuerpo: Contiene expresiones y declaraciones.

Expresiones Lambda: Comparación

Sin lambda

```
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //without lambda,
//Drawable implementation
//using anonymous class
        Drawable d=new Drawable(){
            public void draw(){
     System.out.println("Drawing "+width);}
        };

        d.draw();
    }
}
```

Con lambda

```
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };

        d2.draw();
    }
}
```

Expresiones Lambda: Ejemplo

```
interface Sayable{
    public String say(String name);
}

public class LambdaExpressionExample{
    public static void main(String[] args) {

        // Lambda expression with single parameter.
        Sayable s1=(name)->{
            return "Hello, "+name;
        };
        System.out.println(s1.say("Sonoo"));

        // You can omit function parentheses
        Sayable s2= name ->{
            return "Hello, "+name;
        };
        System.out.println(s2.say("Sonoo"));
    }
}
```

Expresiones Lambda: Ejemplo

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```

Expresiones Lambda: Ejemplo for-each

```
import java.util.*;
public class LambdaExpressionExample{
    public static void main(String[] args) {

        List<String> list=new ArrayList<String>();
        list.add("ankit");
        list.add("mayank");
        list.add("irfan");
        list.add("jai");

        list.forEach(
            (n)->System.out.println(n)
        );
    }
}
```

Expresiones Lambda: Ejemplo

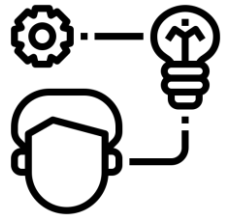
```
@FunctionalInterface
interface Sayable{
    String say(String message);
}

public class LambdaExpressionExample{
    public static void main(String[] args) {

        // You can pass multiple statements in lambda expression

        Sayable person = (message)-> {
            String str1 = "I would like to say, ";
            String str2 = str1 + message;
            return str2;
        };
            System.out.println(person.say("time is precious."));

    }
}
```

Expresiones Lambda: Ejemplo comparador

```
1.import java.util.ArrayList;
2.import java.util.Collections;
3.import java.util.List;
4.class Product{
5.    int id;
6.    String name;
7.    float price;
8.    public Product(int id, String name, float price)
{
9.        super();
10.        this.id = id;
11.        this.name = name;
12.        this.price = price;
13.    }
14.}
```

```
public class LambdaExpressionExample{
    public static void main(String[] args) {
        List<Product> list=new ArrayList<Product>();

        //Adding Products
        list.add(new Product(1,"HP Laptop",25000f));
        list.add(new Product(3,"Keyboard",300f));
        list.add(new Product(2,"Dell Mouse",150f));

        System.out.println("Sorting on the basis of name..
.");

        // implementing lambda expression
        Collections.sort(list,(p1,p2)->{
        return p1.name.compareTo(p2.name);
        });
        for(Product p:list){
            System.out.println(p.id+" "+p.name+" "+p.price
);
        }

    }
}
```

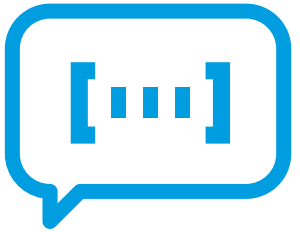
Pongámoslo en práctica



- Usa expresiones lambda para determinar cuál de los coches de una lista es el más rápido en recorrer los 100km

netmind
a bts company

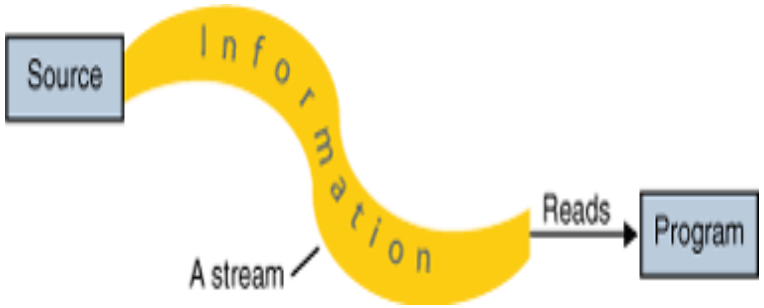
13



STREAMS DE E/S

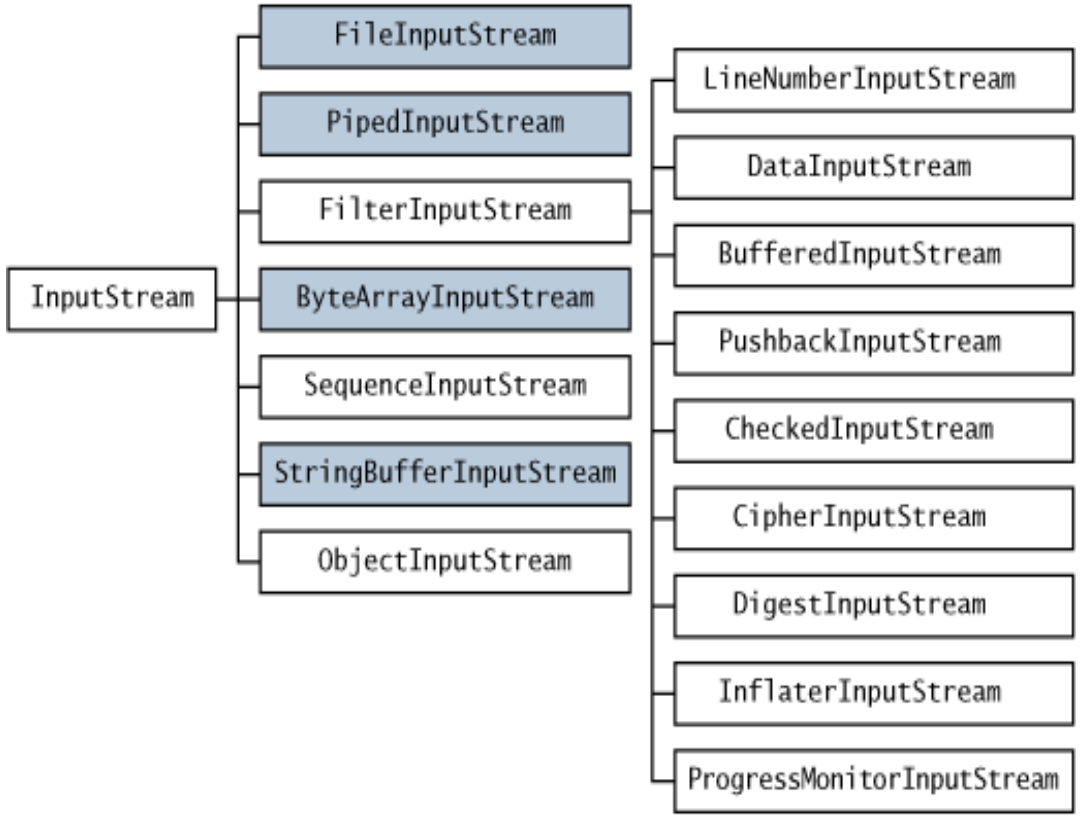
Streams

- Un stream representa un flujo de información:
  - procedente de una fuente (teclado, file, memoria, red, etc.) o
  - dirigida a un destino (pantalla, file, etc.)
- Los streams comparten una misma interfaz que hace abstracción de los detalles específicos de cada dispositivo de E/S.
- Todas las clases de streams están en el paquete **java.io**
  - https://docs.oracle.com/javase/11/docs/api/java/io/package-summary.html



Clases de Streams de entrada

- Los módulos sombreados representan fuentes de datos.
- Los módulos sin sombrear representan procesadores.
- Los procesadores se pueden aplicar a otro procesador o a una fuente.

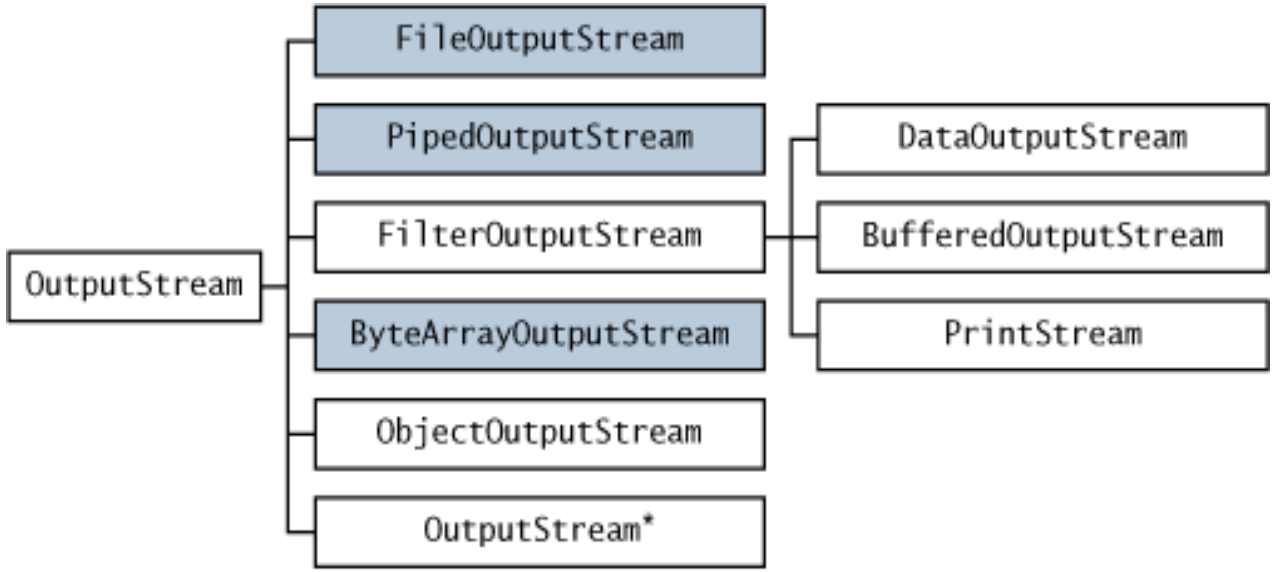


Subclases de InputStream

- FileInputStream: lectura de files byte a byte
- ObjectInputStream: lectura de files con objetos.
- FilterInputStream:
  - BufferedInputStream: lectura con buffer, más eficiente.
  - DataInputStream: lectura de tipos de datos primitivos (int, double, etc.).

Clases de Streams de salida

- Los módulos sombreados representan fuentes de datos.
- Los módulos sin sombrear representan procesadores.
- Los procesadores se pueden aplicar a otro procesador o a una fuente.

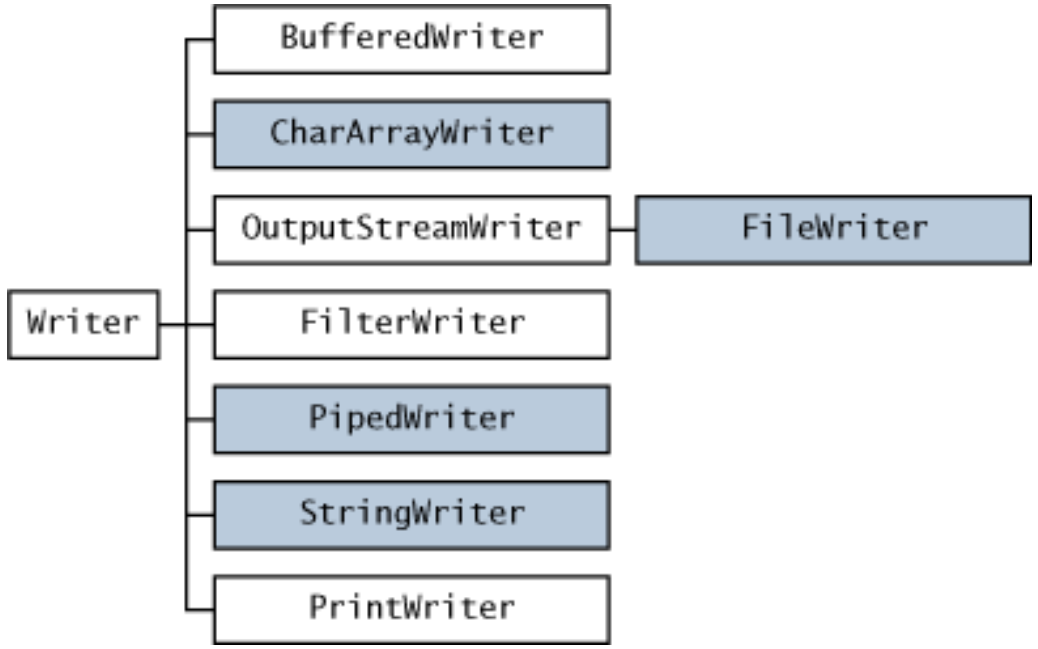


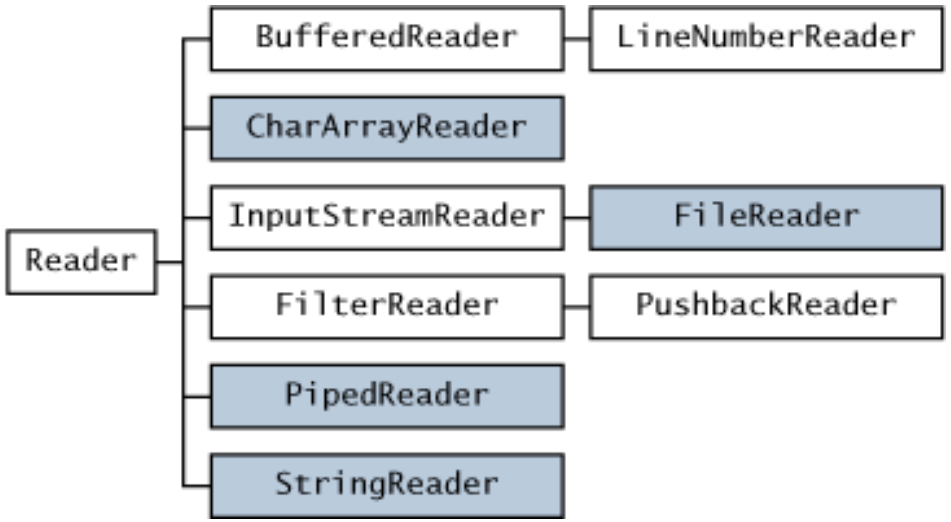
Subclases de OutputStream

- FileOutputStream: escritura de files byte a byte
- ObjectOutputStream: escritura de files con objetos.
- FilterOutputStream:
  - BufferedOutputStream: escritura con buffer, más eficiente.
  - DataOutputStream: escritura de tipos de datos primitivos (int, double, etc.).

Clases de Streams

- Soportan UNICODE (16 bits para un char).
- Módulos sombreados son fuentes, y sin sombrear son procesadores.





Subclases de Reader

- InputStreamReader: convierte un stream de bytes en un stream de chars.
  - FileReader: se asocia a files de chars para leerlos.
- BufferedReader: proporciona entrada de caracteres a través de un buffer (más eficiencia).

Subclases de Writer

- OutputStreamWriter: convierte un stream de bytes en un stream de chars.
  - FileWriter: se asocia a files de chars para modificarlos.
- BufferedWriter: proporciona salida de caracteres a través de un buffer (más eficiencia).
- PrintWriter: métodos print() y println() para distintos tipos de datos.

Otras Clases de java.io

- File: permite realizar operaciones habituales con files y directorios.
- RandomAccessFile: permite acceder al n-ésimo registro de un file sin pasar por los anteriores.
- StreamTokenizer: permite “trocear” un stream en tokens.

Ejemplo Streams - lectura por líneas

```
public static void main(String[] args) throws IOException {
     // 1a. Se lee un file línea a línea
     BufferedReader in = new BufferedReader(new FileReader("IOStreamDemo.java"));
     String s, s2 = new String();
     while( (s = in.readLine()) != null) s2 += s + "\n";
     in.close();
     // 1b. Se lee una línea por teclado
     BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
     System.out.print("Enter a line:"); System.out.println(stdin.readLine());
}
```

Parsing de tipos básicos

```
String linea;
int a;
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
System.out.print("Enter a line:");
linea = stdin.readLine();
a = Integer.parseInt(linea);
System.out.println(a);
```

Ejemplo Streams - escritura por líneas

```
// throws IOException
String []s = {“hola“, “que”, “tal”};
// Se inicializa s
PrintWriter out1 = new PrintWriter(new BufferedWriter( new FileWriter("IODemo.out")));
int lineCount = 1;
for (int i=0; i<s.length(); i++) out1.println(lineCount++ + ": " + s[i]);
out1.close();
```

Ejemplos Streams- escritura de tipos básicos

```
// throws IOException
DataOutputStream out2 =
new  DataOutputStream(new BufferedOutputStream(new FileOutputStream("Data.txt")));

out2.writeDouble(3.14159);
out2.writeBytes("That was pi\n“);
out2.writeChars( "That was pi\n“);
out2.writeDouble(1.41413);
out2.writeUTF("Square root of 2");
out2.close();
```

Ejemplo Streams - lectura de tipos básicos

```
// throws IOException
DataInputStream in5 =
    new DataInputStream( new BufferedInputStream(new FileInputStream("Data.txt")));

System.out.println(in5.readDouble());
System.out.println(in5.readLine()); // deprecated
System.out.println(in5.readDouble());
System.out.println(in5.readUTF());
```

Ejemplo Streams - files de acceso aleatorio

```
// throws IOException
 // acceso de lectura/escritura
RandomAccessFile rf = new RandomAccessFile("rtest.dat", "rw");
for(int i = 0; i < 10; i++) rf.writeDouble(i*1.00);
rf.close();

rf = new RandomAccessFile("rtest.dat", "rw");
rf.seek(5*8); // salta 5 doubles (8 bytes cada uno)
rf.writeDouble(47.00); // modifica el sexto double
rf.close();
 // acceso de sólo lectura
rf = new RandomAccessFile("rtest.dat", "r");
for(int i = 0; i < 10; i++) {
    System.out.println( "Value " + i + ": " + rf.readDouble()); rf.close();
}
```

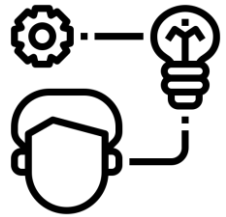
Ejemplo Streams - Object Stream

- Java permite guardar objetos en ficheros, pero esos objetos deben implementar la interfaz Serializable:

```
public class MySerializableClass implements Serializable { ... }
```

```
// throws IOException
FileOutputStream out = new FileOutputStream("theTime.dat");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date(1,1,2006));
s.close();

// throws IOException y ClassNotFoundException
FileInputStream in = new FileInputStream("theTime.dat");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
s.close();
```

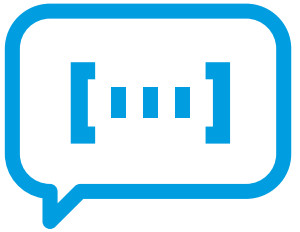
Pongámoslo en práctica



- Queremos dejar las características de nuevos coches en distintos ficheros y que el programa principal los lea y genere con ellos la lista a comparar
- Asimismo, una vez comparado, escriba en un nuevo fichero “resultados.txt” el ranking de velocidades

netmind
a bts company

14



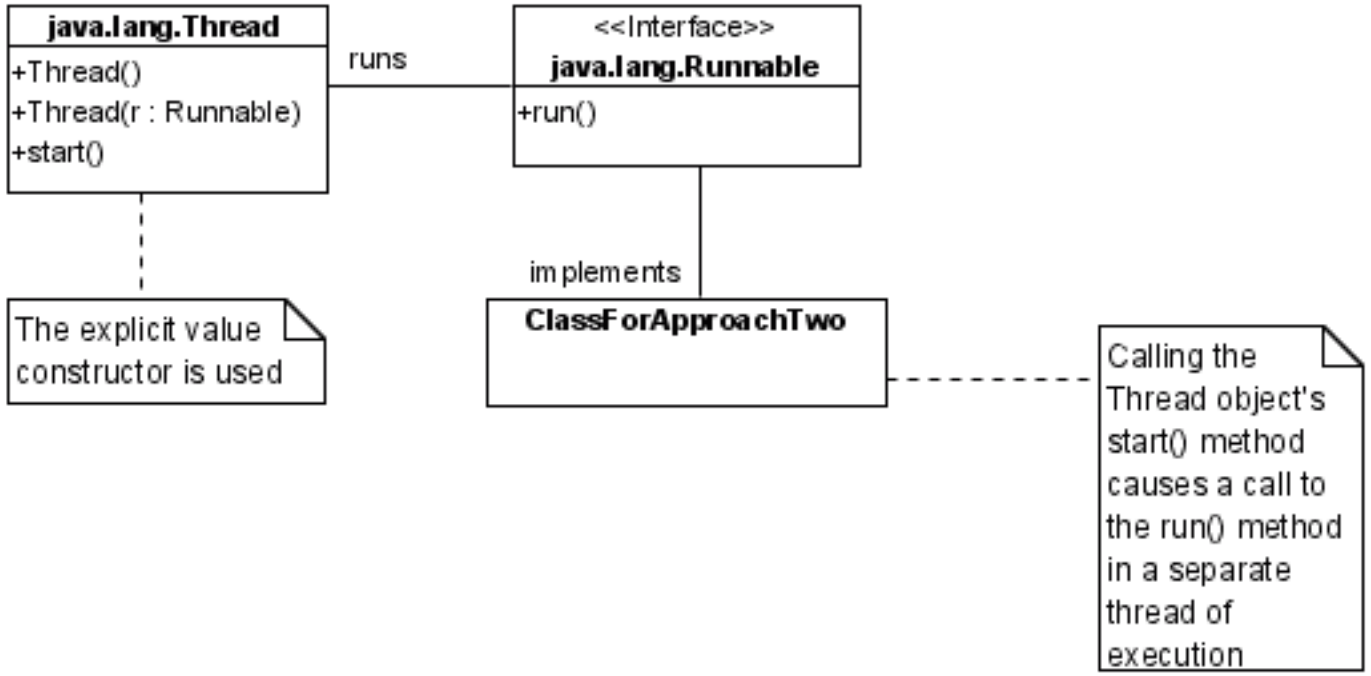
APLICACIONES MULTITHREAD

Threads

- Crear un thread en Java es una tarea sencilla.
- Es un mecanismo que depende de la plataforma sobre la que se esté ejecutando la JVM.
- Un programa basado en threads puede realizar varias tareas de forma concurrente (Al menos ésa es la impresión que recibe el usuario, cuando el sistema cuanta con 1 sola CPU).
- Actualmente los sistemas operativos son multitarea, existen básicamente
- dos tipos:
  - Basada en procesos.
  - Basada en hilos de ejecución o threads.

Threads en Java

- Existen dos formas de trabajar con Threads en Java, extendiendo de la clase Thread o implementando el interface Runnable:



La clase Thread

- Podemos crear threads a partir de la clase Thread y sobre escribiendo el método run(), que será invocado automáticamente en el momento que pase al estado de runnable. Podemos usar métodos como join() o notify() para sincronizar procesos:

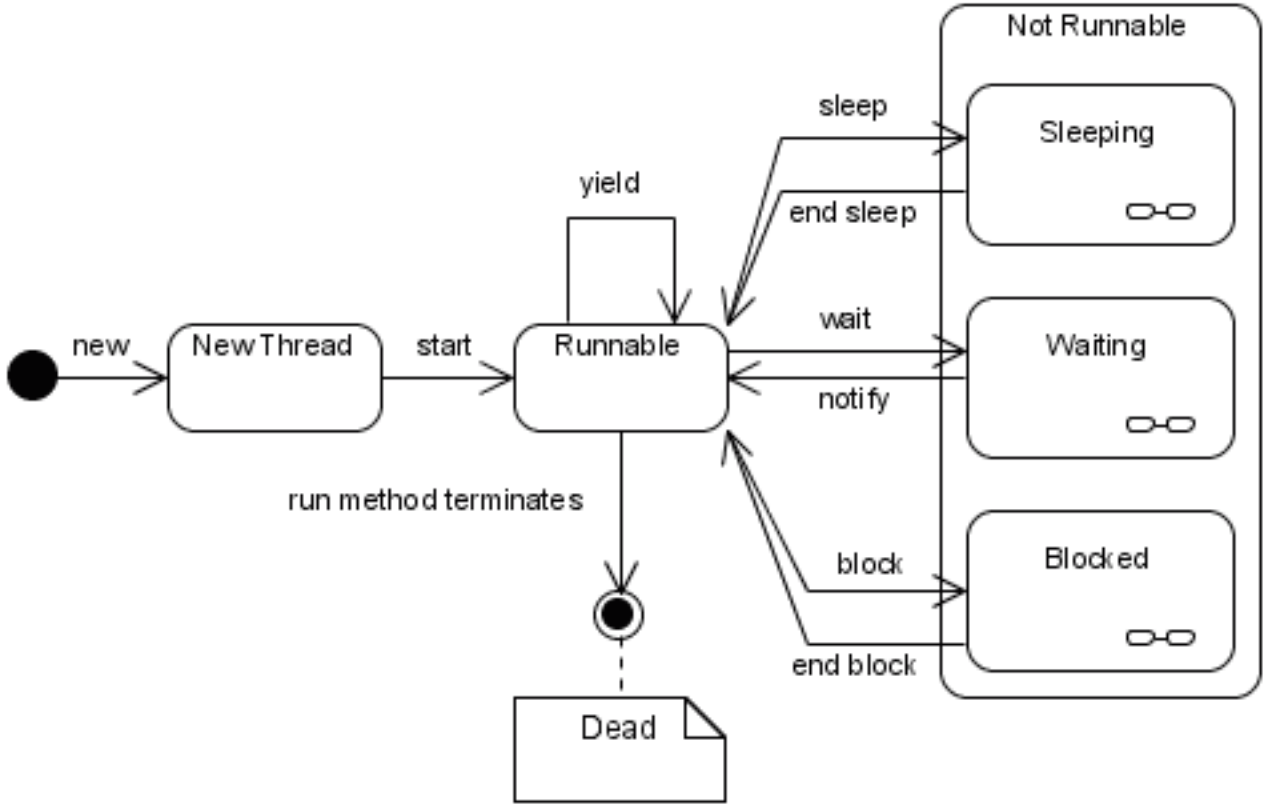
Método / Propiedad	Descripción
MAX_PRIORITY	Máxima prioridad que puede tener un hilo (thread)
MIN_PRIORITY	Mínima prioridad que puede tener un hilo (thread)
NORM_PRIORITY	Prioridad predeterminada que se asigna a un hilo (thread)
activeCount()	Devuelve el número de hilos activos en el grupo de hilos del hilo actual
currentThread()	Devuelve una referencia al objeto que representa al hilo en ejecución
getID()	Devuelve el identificador del hilo actual (thread)
getName()	Devuelve el nombre del hilo
getPriority()	Devuelve la prioridad del hilo
getThreadGroup()	Devuelve el grupo de hilos al que pertenece el hilo
getState()	Devuelve el estado del hilo actual
isAlive()	Comprueba si el hilo está vivo

La clase Thread

Método / Propiedad	Descripción
isDaemon()	Comprueba si el hilo es un hilo daemon
isInterrupted()	Comprueba si el hilo ha sido interrumpido
join()	Espera a que muera el hilo
run()	Si el hilo utiliza un objeto Runnable è se ejecuta este método
setName()	Cambia el nombre del hilo
setPriority()	Cambia la prioridad del hilo
sleep()	Indica al hilo que se interrumpa durante los milisegundos indicados
start()	Indica al hilo que comience la ejecución, la JVM llama al método run de este
hilo	Indica al hilo en ejecución que se pause temporalmente y permitir que entre
yield()	Marca el hilo como hilo daemon o user
otro hilo	Comprueba si el hilo es un hilo daemon

Ciclo de vida de un Thread

- La JVM dispone de un mecanismo llamado schedule para trabajar con los threads sobre cualquier plataforma, la cual controla el ciclo de vida de un thread en Java



Ciclo de vida de un Thread

- Java proporciona una forma eficiente en la que varios threads pueden comunicarse.
- Esto reduce el tiempo de espera de la cpu, para ello existen varios métodos que permiten llevar a cabo esta tarea. Deben estar situados en un bloque sincronizado:

Método / Propiedad	Descripción
wait()	Indica que el hilo que realiza la llamada espera hasta que que otros hilos comiencen a monitorizar utilizando los métodos notify() o notifyAll()
notify()	Despierta al primer hilo que llame al método wait()
notyfyAll()	Despierta a todos los hilos que llamaron al método wait(). El hilo con mayor prioridad se ejecutará primero.

- Estos métodos deben estar encerrados en un bloque try/catch.
- Estos métodos trabajan con la agenda (schedule) de threads

Evitar los deadlocks

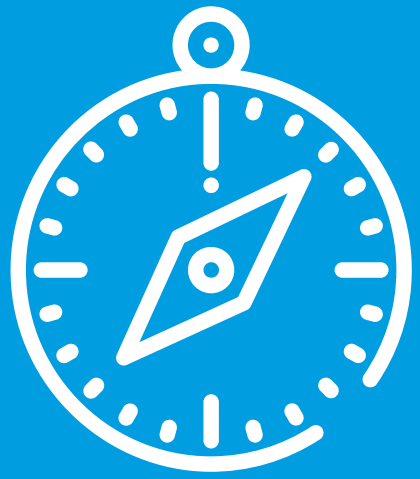
- Para evitar que varios threads trabajen de forma simultanea sobre el mismo código o recurso, se hace necesario utilizar la sincronización de código en Java.
- Cada objeto en Java dispone de un bloqueo, podemos actuar sobre el para obtener un acceso exclusivo para un thread de forma concurrente.
- Cuando un thread entra en un bloque synchronized adquiere el bloqueo del objeto y no lo libera hasta haber finalizado su tarea:

```
synchronized (object) {
    statement block;
}
```

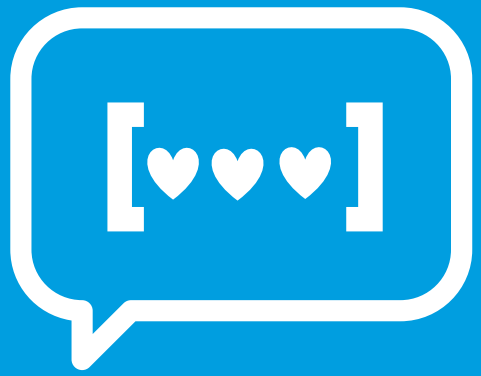
Garbage collector

- En Java no es necesario reservar zonas de memoria ni utilizar punteros para gestionar el mapa de memoria, por esta razón, tampoco existe la necesidad de liberar estos recursos.
- El recolector de basura lo hace por nosotros. El GC arranca cuando no existen referencias a un objeto y libera recursos.
- Básicamente lleva a cabo tres tareas:
  - 1.Monitoriza objetos para conocer cuando dejan de usarse.
  - 2.Informa de los recursos que pueden liberar.
  - 3.Reclama estos recursos una vez destruido el objeto.

- Trabaja en un Thread separado en el background para controlar a todos los objetos. Un objeto “existe”, mientras exista una referencia a el.



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

netmind
a bts company

Thanks!

Follow us: