

REPÚBLICA BOLIVARIANA DE VENEZUELA

MINISTERIO DEL PODER POPULAR PARA LA EDUCACIÓN UNIVERSITARIA

INSTITUTO UNIVERSITARIO POLITÉCNICO DE LOS LLANOS “JUANA RAMIREZ  
LA AVANZADORA”

PROGRAMA NACIONAL DE FORMACIÓN EN INFORMÁTICA

UNIDAD CURRICULAR: PROGRAMACIÓN Y ALGORITMÍA

SECCIÓN 2, TRAYECTO II

# **PROGRAMA DE LAS 8 REINAS USANDO LA TÉCNICA DE BACKTRACKING EN LA MÉTODOLOGÍA DE LA PROGRAMACIÓN ORIENTADA A OBJETOS**

**FACILITADOR:**

PROF. CARLOS PADRÓN

**INTEGRANTES:**

RENGIFO LIBARDO C.I: 2800010320

DICIEMBRE 2020

## Introducción

El problema fue originalmente propuesto en 1848 por el ajedrecista Max Bezzel. Durante años, muchos matemáticos, incluyendo a Gauss y a Georg Cantor, han trabajado en él y lo han generalizado a  $n$ -reinas. Las primeras soluciones fueron ofrecidas por Franz Nauck en 1850. Nauck también se abocó a las  $n$ -reinas (en un tablero de  $n \times n$  de tamaño arbitrario). En 1874, S. Günther propuso un método para hallar las soluciones usando determinantes, y J.W.L. Glaisher redefinió su aproximación.

Edsger Dijkstra usó este problema en 1972 para ilustrar el poder de la llamada programación estructurada. Publicó una descripción muy detallada del desarrollo del algoritmo de backtracking, "depth-first".

El algoritmo planteado a continuación prone una solución con backtracking recursivamente usando una pila dinámica, la cual se ira llenando con los indices de las columnas a las cuales las reinas fueron colocadas, cada posición de la pila representa una fila dentro del tablero.

Con ello a través de unas meticulosos y sencillos cáculos mateáticos se puede comprobar la seguridad de una casilla dentro de la pila, así mismo se logra plantear la solución al problema utilizando la metodología de la prograamación orientada a objetos.

## Puzzle de las 8 reinas

El **problema de las ocho reinas** es un pasatiempo que consiste en poner ocho reinas en el tablero de ajedrez sin que se amenacen. Fue propuesto por el ajedrecista alemán Max Bezzel en 1848.

### Algoritmo

El algoritmo consiste en usar una pila dinámica de enteros, la cual se van a introducir los índices de las columnas de donde se encuentran cada reina dentro de un arreglo de enteros.

La pila se ira llenando de acuerdo a ciertos criterios, la reina no debe ser amenazada por las columnas, filas y diagonales.

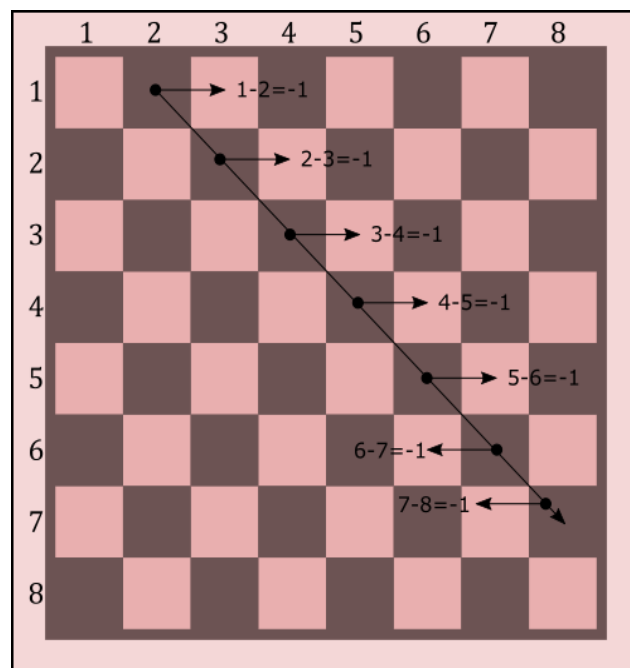
Para ello se realiza una comprobación sencilla para verificar las diagonales, se sigue la siguiente regla:

“Con esto en mente, se puede comprobar que en una misma diagonal descendente, todas las casillas cumplen que la diferencia entre el número de fila y el número de columna es constante:  $i - j = k$ . Siendo  $i$  el número de fila y  $j$  el número de columna.”

Para facilitar la comprensión del criterio

mencionado, se usa el valor absoluto de la resta de  $i - j = k$ , de esta forma se trabaja más fácil con dichas operaciones.

En el caso de las columnas, se comprueba por medio de un ciclo que itera hasta llegar al tamaño de la pila (con el objetivo de disminuir el rango de comprobacion de columnas), si



la variable contadora (que representa una columna) es igual al índice de la columna almacenado en el arreglo entonces dicha columna esta ocupada por otra reina.

Si algunos de los 2 criterios mencionados anteriormente se cumplen, entonces no es posible colocar una reina en esa fila y columna.

El caso base para detenerla recursión del algoritmo, es aquella cuando la pila llegase a estar completamente vacía (**null**) y en dicha instancia se elimina las referencias de los objetos creados (pila y arreglo).

Si la pila llegase a estar llena es decir el tamaño de la pila es igual a la cantidad de reinas a buscar (**pila.size() == cantidadReinas**) , quiere decir que se encontro una solución sastifactoria.

Posteriormente de aplicar los pasos anteriores, se procede a eliminar la ultima reina en la pila y correrla una columna a la derecha, con el objetivo de seguir comprobando, y así que en algún punto, la pila llegue a estar nula (ya que la posicion de la primera reina no sastiface la colocación de las demás) y acabarse todas las soluciones posibles.

Para finalizar, se vuelve a invocar la función con los parámetros ya modificados y buscar otra solución que sastiface los criterios mencionados.

## **Implementación en el lenguaje java (clases)**

El programa costa de una serie de clases y métodos para lograr la solución del problema, a continuación se hace mención de ellos con una breve explicación de su uso y sus métodos:

### **Acciones.java**

Interfaz de la la cual las clases: Programa y Algoritmo heredan. Esta interfaz sirve para dar soporte a la 2 clases mencionadas ya que comparten un método en común, el de iniciar su algoritmo.

### **Helpers.java**

Es una clase ayudante para facilitar el trabajo de algunas operaciones, como las siguientes:

- **imprimir(Object o)** se usa para imprimir en consola un objeto.
- **imprimirEnLinea(Object o)** se usa para imprimir sin salto de línea.
- **esNumero(String str)** se usa para saber si una cadena es un número.
- **numeroDentroDelRango (int n)** se usa para saber si  $n \geq 1$  y  $n \leq 8$ .
- **leerCadena()** se usa para leer una cadena desde el teclado.

## Programa.java

Clase que se utiliza para pedir datos, en este caso la cantidad de reinas que se van a usar dentro del tablero. Esta clase además, implementa un método heredado definido en la interfaz Acciones, dicho método tiene la lógica para pedir la cantidad de reinas y además esta comprobado para que dicha cantidad sea un valor numerico válido y comprendido entre 1-8.

## Algoritmo.java

Es una clase la cual busca las soluciones de las 8 reinas por medio de una serie de métodos, los cuales son los siguientes:

- **comenzar()** iniciar la búsqueda de soluciones.
- **resolver(int columnaActual, int solucionesEncontradas, piladinámica <Integer> pila)** método recursivo que busca las soluciones, necesita la columna en la que va iniciar, sus soluciones encontradas hasta ahora y la pila.

- **esCorrectoElTablero(pila dinámica <Integer> pila, int columnaActual, int tablero[])** método que indica si una columna dada es adecuada para poner una reina, recibe la pila y el tablero, y por medio de un calculo matematico sencillo, se busca si dicha posicion es correcta.
- **imprimirSolucion(int solucion)** imprime el tablero con la solucion dada.
- 

## Reinas.java

Clase principal la cual el programa se ejecuta. Dicha clase ejecuta el método main el cual instancia un nuevo objeto aplicando el polimorfismo con la interfaz acciones, dicha instancia posee un método para iniciar el programa.

## Explicación del código de las 8 reinas.

### Acciones.java

Para empezar se inicia por la interfaz de Acciones, esta define un método void llamado comenzar(), dicho método será heredado en las clases Programa.java y Algoritmo.java (se sobrescribe @Override) en las clases mencionadas, se muestra una imagen de la interfaz:

```
package pkg8.reinas.pilas;
public interface Acciones {

    public void comenzar();
}
```

El método comenzar() se hereda en las clases Programa.java y Algoritmo.java  
De esa forma se realiza ligadura dinámica de métodos

De esta manera se aplicará **polimorfismo** por interfaz y también **ligadura dinámica de métodos**, es decir, el lenguaje java detectará la implementación del método comenzar() dinámicamente, de acuerdo a cual clase se esté instanceando e invocando el método.

## Reinas.java

Reinas.java es la clase la cual inicia el programa a través de la instanciación de un objeto de la clase llamada Programa.java, la cual hereda de la interfaz Acciones.java.

De esta forma se puede sobrescribir su método comenzar() para que la clase Programa.java ejecute su propia implementación, por lo tanto se estaría realizando 2 características de la Programación Orientada a Objetos: **Polimorfismo** con interfaces y **ligadura dinámica de métodos**.

```
package pkg8.reinas.pilas;

import static pkg8.reinas.pilas.Helpers.imprimir;

public class Reinas {


    public static void main(String[] args) {
        imprimir("Algoritmo de las 8 reinas");
        imprimir("Coloque la cantidad de reinas");

        Acciones p = new Programa();
        p.comenzar(); // ligadura dinamica de metodos
        p = null;
    }
}
```

```
package pkg8.reinas.pilas;

public interface Acciones {

    public void comenzar();
}
```



Se puede observar la clase Programa.java herada de la interfaz Acciones, por lo tanto se tiene la seguridad de que la clase Programa implementa el método comenzar(), por lo tanto se invoca y se ejecuta del código que esta escrito en la clase Programa, este es un tipo de polimorfismo con interfaces lo cual permite ligadura de métodos a través de la herencia.

Después de que el programa termine su ejecución, el objeto **p** será destruido de la memoria haciendolo que apunte a **NULL**.

## Programa.java

Es una clase que representa el programa en sí, contiene comprobación de tipos y rangos en la entrada de la cantidad de reinas a buscar (número positivo 1-8), esta clase hereda de la interfaz Acciones.java, por lo tanto debe implementar y sobrescribir el método heredado comenzar(), de esta forma se garantiza que la **ligadura dinámica de métodos** se haga presente y ejecute el código correspondiente, la clase Programa.java también instancia un objeto de la clase Algoritmo.java, la cual también hereda de la interfaz Acciones.java, lo cual hace disponible el método comenzar(), de esta forma se ejecuta la implementación creada en la clase Algoritmo.java, dicha clase posee un **constructor** mediante el cual se recibe la cantidad de reinas a buscar.

```
package pkg8.reinas.pilas;

import static pkg8.reinas.pilas.Helpers.esNumero;
import static pkg8.reinas.pilas.Helpers.imprimir;
import static pkg8.reinas.pilas.Helpers.leerCadena;
import static pkg8.reinas.pilas.Helpers.numeroDentroDelRango;

public class Programa implements Acciones {
    @Override
    public void comenzar() {
        boolean seguir = true;

        while (seguir) {
            String str = leerCadena();
            if (!str.isEmpty() && esNumero(str)) {
                int reinas = Integer.parseInt(str);
                if (numeroDentroDelRango(reinas)) {
                    Acciones ag = new Algoritmo(reinas); // constructor
                    ag.comenzar(); // ligadura dinamica de metodos
                    ag = null;
                    seguir = false;
                } else {
                    imprimir("El valor ingresado no es un numero en el rango 1-8.\nIntente de nuevo:");
                }
            } else {
                imprimir("El valor ingresado no es un numero valido.\nIntente de nuevo:");
            }
        }
    }
}
```

Clases ayudantes o auxiliares del programa

Sobreescritura de métodos

La clase Programa implementa el método comenzar() de la interfaz Acciones, por ende se aplica el polimorfismo y la ligadura dinámica de metodos ya que se esta ejecutando código de una clase especifica (Programa).

Algoritmo tiene un constructor, que toma la cantidad de reinas a buscar

```
package pkg8.reinas.pilas;
public interface Acciones {
    public void comenzar();
}
```

Despues se instancia un objeo de la clase Algoritmo la cual tambien hereda de la interfaz Acciones, por ende también implementa el método comenzar() a traves de la herencia, por ende también se aplica la ligadura dinámica de métodos que se logra a través del Polimorfismo de interfaces



Después de realizar todos los pasos anteriores, se destruye el objeto **ag** que es la instancia de la clase `Algoritmo.java`, de esta forma se garantiza que se elimina de la memoria, ya que apunta a **NULL**.

## Algoritmo.java

Esta clase es la más extensa y se dividirá en diferentes explicaciones, esta clase se encarga de buscar las soluciones por medio de **backtracking** de forma **recursiva** usando una **pila dinámica** y un **arreglo** de números naturales.

Dicha clase posee un **constructor** que recibe la cantidad de reinas que el algoritmo va a buscar, además que hereda de la interfaz `Acciones.java`, la cual implementa el método `comenzar()`, que más adelante se explicará su uso.

```
package pkg8.reinas.pilas;

import pkg8.reinas.Stack.piladinamica;
import static pkg8.reinas.pilas.Helpers.imprimirEnLinea;
import static pkg8.reinas.pilas.Helpers.imprimir;

public class Algoritmo implements Acciones{

    int cantidadReinas;
    int tablero[];

    // constructor
    public Algoritmo(int reinas) {
        this.cantidadReinas = reinas;
        this.tablero = new int[reinas];
    }
```

Clases ayudantes o auxiliares

Se hereda de la interfaz  
Acciones

Arreglo con las columnas de las reinas

Constructor de la clase, recibe cuantas reinas se van a poner en el  
tablero

Inicialización de propiedades de las clases

Luego se tiene el método `comenzar`, su función es iniciar la búsqueda de las reinas en el tablero por medio del método `resolver()`, mediante una **pila** que contiene las columnas de las reinas, cabe destacar que este método `comenzar` se sobrescribe para que la **ligadura dinámica** de métodos pueda surgir y no haya errores de referencias, por ello se usa **@Override**.

```

@Override
public void comenzar() {
    piladinamica<Integer> pila = new piladinamica<>();
    resolver(0, 0, pila); // usando la pila dinamica
    pila = null;
}

```

Se sobrescribe el método comenzar de la interfaz Acciones.java

Se procede a realizar la implementación del método comenzar()

Se elimina el objeto creado, en este caso la pila

Luego de la búsqueda se elimina la **pila** en la memoria.

Posteriormente se define el método resolver(), que de manera **recursiva** y aplicando **backtracking** se logra encontrar las reinas.

Método recursivo, que toma la columna en la que se esta iterando actualmente, las soluciones que se han encontrado, y la pila la cual tiene las columnas de las reinas que en ese momento se han colocado en el tablero

```

public void resolver(int columnaActual, int solucionesEncontradas, piladinamica<Integer> pila) {
    if (cantidadReinas == 2 | cantidadReinas == 3) {
        imprimir("Para esta cantidad no existen soluciones.");
        return;
    }
}

```

Para tener un mejor rendimiento, se evita hacer invocaciones recursivas para los casos de 2 o 3 reinas, ya que como se sabe no existe soluciones para tal cantidad de reinas.

Posterior, se procede a buscar columnas disponibles para colocar reinas dentro de la **pila** y del **arreglo**, para ello se emplea el método esCorrectoElTablero() para saber si en la columna actual y el tablero con la pila son posiciones con reinas a las cuales dicha columna no amenazan, si es verdad solo se empujan la columna en la pila y en el arreglo, de lo contrario se pasa a la siguiente columna.

Si en el tablero (arreglo tablero) con la columna "columnaActual" y con la reinas ya colocadas (en la pila), se puede poner una reina en dicha columna, entonces se procede a empujarla (su columna) dentro de la pila y también en el arreglo, luego se procede a comprobar desde la prima columna, para determinar si no existen casillas disponibles

```
while (columnaActual < cantidadReinas) {  
  if (esCorrectoElTablero(pila, columnaActual, tablero)) {  
    pila.push(columnaActual);  
    tablero[pila.size() - 1] = columnaActual;  
    columnaActual = 0;  
  } else {  
    columnaActual++;  
  }  
}
```

Se empujan las columnas (reinas) dentro de la pila y el arreglo

Si no es posible colocar una reina en la columna "columnaActual" entonces se procede a pasar a la siguiente columna, para seguir buscando mas casillas disponibles

Siguiendo con el flujo del programa, cuando se sale del ciclo while, es por la siguientes razones, se encontró una solución de N soluciones existentes o la pila esta vacía (**NULL**).

Cuando se sale del ciclo while, se comprueba si el tamaño de la pila es igual a la cantidad de reinas que se desea buscar, si es así entonces se encontró una solución, se aumenta el contador de soluciones y se imprime la solución en pantalla

```
// si la pila tiene 8 elementos (reinas), eso quiere decir que se encontro una solucion  
if (pila.size() == cantidadReinas) {  
  solucionesEncontradas++;  
  imprimirSolucion(solucionesEncontradas);  
}
```

El otro caso (el caso base de la recursión), podría darse cuando la pila este vacía es decir **NULL** por lo tanto ya todas las soluciones fueron encontradas, ya que mínimo, siempre estará una reina en el tablero, a excepción de la primera iteración del programa.

Ya no existen más soluciones, porque no hay una reina en el tope de la pila, por lo tanto el tablero está vacío.

```
if (pila.top() == null) {  
    pila = null;  
    tablero = null;  
    return;  
}
```

Se destruye las referencias en la memoria

Al final del método ocurre la técnica del **backtracking**, si los casos anteriores no se dan, entonces se retrocede a la última reina y se pasa a la siguiente columna, y luego se vuelve a buscar otras soluciones.

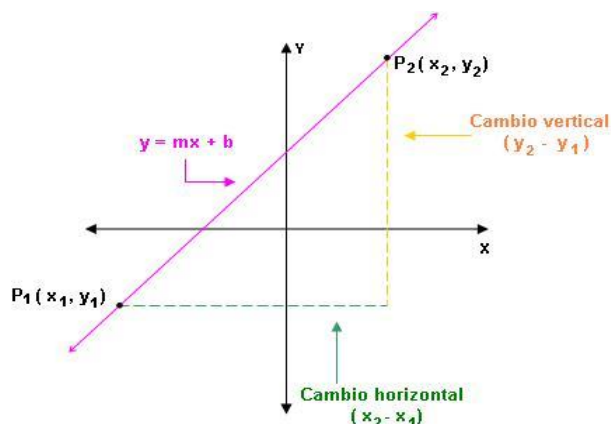
Aquí ocurre el **BACKTRACKING** cuando el caso base no ocurre, por lo tanto se procede a eliminar la última dama colocada, se toma su columna y se le suma una unidad, para comprobar la siguiente columna, de esta forma siempre se retrocede y se comprueba en la anterior columna.

```
columnaActual = pila.pop() + 1;
```

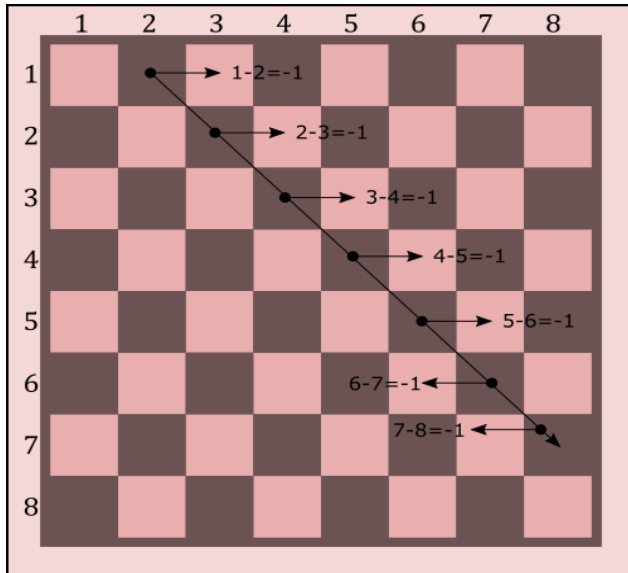
```
// volvemos a llamar con las variables ya modificadas  
resolver(columnaActual, solucionesEncontradas, pila);
```

Llamada recursiva, ya los parametros cambiarón su valor, por ende se evita una recursión infinita

Otro método que usa la clase Algoritmo.java, es el método esCorrectoElTablero(), el cual dadas 2 reinas (**puntos**) dentro del tablero (**plano cartesiano**) verifica por columnas y diagonalmente a través de una formula matemática sencilla, la cual es: determinar la pendiente entre 2 puntos, en este caso de reinas, un ejemplo de la formula es la siguiente:



Es decir en la gráfica anterior  $P_1$  sería una reina ubicada en  $X_1$  (fila) y  $Y_1$  (columna), de igual forma el punto  $P_2$  que sería la otra reina, como puede observar todo se trabaja a través de coordenadas en un plano cartesiano.



Ahora, la formula anterior se utilizó ya que las diagonales cumplen con un **criterio**, sus sumas o restas de sus filas y columnas siempre serán constantes, ya que uno de sus ejes siempre aumentará o disminuirá dependiendo de cual sera la dirección de la diagonal.

Como puede observarse en la imagen.

Lo que se traduce a la siguiente formula para aplicarla en el algoritmo:

$$|X_1 - X_2| = |Y_1 - Y_2|$$

La implementación en java para dicha formula sería de esta forma:

```
boolean esCorrectoElTablero(piladinamica<Integer> pila, int columnaActual, int tablero[]) {
```

```
    for (int columna = 0; columna < pila.size(); columna++) {
```

```
        boolean mismaColumna = tablero[columna] == columnaActual; // Si la columna pasada por parametro es igual a la columna que está dentro de la pila, entonces estan en la misma columna
```

Como se explico anteriormente, se aplica la formula matemática para calcular la pendiente entre 2 puntos (reinas):  $|X_1 - X_2| = |Y_1 - Y_2|$

```
        boolean mismaDiagonal = Math.abs(columnaActual - tablero[columna]) == pila.size() - columna;
```

Si se amenazan por columna o diagonal, no es posible poner la reina

```
        if (mismaColumna || mismaDiagonal) {
            return false;
        }
```

```
    }
    return true;
}
```

No es necesario sacar el valor absoluto, ya que la variable "columna" siempre será menor al tamaño de la pila

De esta manera se aprovecha el criterio de las columnas y de las filas de un tablero de ajedrez, una característica muy curiosa.

Posteriormente se tiene el método para imprimir las reinas dentro del tablero, la lógica es simple, se va iterando por cada posición del arreglo comprobando si existe una reina en dicha fila.

```
void imprimirSolucion(int solucion) {
    imprimir(solucion + ": ");
    for (int fila = 0; fila < cantidadReinas; fila++) {
        for (int columna = 0; columna < cantidadReinas; columna++) {
            // se va comprobando si en la fila existe una reina
            if (fila == tablero[columna]) {
                imprimirEnLinea("Q ");
            } else {
                imprimirEnLinea("# ");
            }
        }
        imprimirEnLinea("\n");
    }
    imprimir("");
}
```

Se va comprobando si la fila contiene una reina en "columna" posición

Y por ultimo y no menos importante, los métodos de la clase ayudante, para evitar escribir código repetitivo, (siguiente página)

```

public class Helpers {

    public static void imprimir(Object o) { _____ Imprime en pantalla un mensaje con salto de
        System.out.println(o); _____ línea
    }

    public static void imprimirEnLinea(Object o) { _____ Lo mismo que el anterior pero sin salto de línea
        System.out.print(o);
    }

    public static boolean esNumero(String str) { _____ Comprueba si una cadena es un número, intentado
        boolean numeric = true; _____ convertir dicha cadena a un número, si falla, arroja
        _____ una excepción y se maneja para cambiar la
        _____ bandera lógica "numeric"

        try {
            Double num = Double.parseDouble(str);
        } catch (NumberFormatException e) {
            numeric = false;
        }

        return numeric;
    }

    public static boolean numeroDentroDelRango(int n) { _____ Se comprueba en rango del número
        return n >= 1 && n <= 8;
    }

    public static String leerCadena() { _____ Método para leer cadenas desde el teclado
        Scanner sc = new Scanner(System.in);
        String input = sc.nextLine();
        return input;
    }
}

```

## Términos usados en este trabajo

A continuación se da a conocer términos usados en este proyecto, con algunos ejemplos dentro del algoritmo planteado.

### Clase

Una clase es la descripción de un **conjunto de objetos** similares; consta de métodos y de datos que resumen las **características comunes** de dicho conjunto.

Clase principal, la cual inicia el programa

```
public class Reinas {  
    public static void main(String[] args) {  
        imprimir("Algoritmo de las 8 reinas");  
        imprimir("Coloque la cantidad de reinas");  
  
        Acciones p = new Programa();  
        p.comenzar(); // ligadura dinamica de metodos  
        p = null;  
    }  
}
```

## Objeto

Se trata de un ente abstracto usado en programación que permite separar los diferentes componentes de un programa, simplificando así su elaboración, depuración y posteriores mejoras. A los objetos se les otorga ciertas características en la vida real.

```
Acciones p = new Programa();  
p.comenzar(); // ligadura dinan  
p = null;
```

## Método

Son aquellas funciones que permite efectuar el objeto y que nos rinden algún tipo de servicio durante el transcurso del programa.

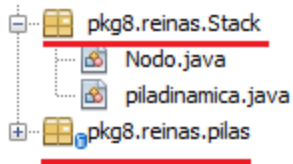
Determinan a su vez como va a responder el objeto cuando recibe una interacción del programa.

```
Acciones p = new Programa();  
p.comenzar(); // ligadura di  
  
p = null;
```



## Empaquetamiento

Este sistema de empaquetamiento se encarga de empaquetar o agrupar clases que se relacionan entre sí, con el objeto de mantener el entorno más organizado.



## Encapsulación

Define el comportamiento de una clase u objeto que tiene dentro de él todo tipo de métodos y datos pero que solo es accesible mediante el paso de mensajes. y los datos a través de los métodos del objeto/clase.

## Constructor

Un "**constructor**" es un método especial que se ejecuta en el momento que se crea un objeto de la clase (**new**). Su misión es reservar memoria e inicializar las variables miembro de la clase.

Se pasan los parametros al constructor de la clase:

```
Acciones ag = new Algoritmo(reinas);
```

Luego se reciben los parámetros en la clase:

```

public Algoritmo(int reinas) {
    this.cantidadReinas = reinas;
    this.tablero = new int[reinadas];
}

```

## Destructor

La misión más común de los destructores es liberar la memoria asignada por los constructores, aunque también puede consistir en desasignar y/o liberar determinados recursos asignados por estos.

## Polimorfismo

El polimorfismo es un sistema de tipos, de tal manera que una referencia a una clase acepta direcciones de objetos de dicha clase y de sus clases derivadas

```

Acciones ag = new Algoritmo(reinas);
ag.comenzar();
ag = null;
seguir = false;

```

## Instanciación

Es la capacidad de crear nuevos objetos a partir de una clase, esta instanciación se hace con el operador **new**, acto seguido el nombre de la clase, y parámetros si la clase posee constructor.

```

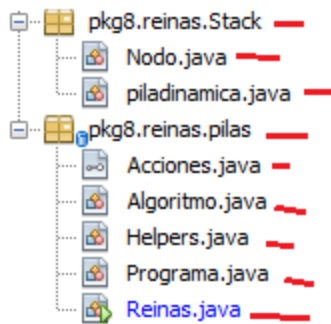
@Override
public void comenzar() {
    piladinamica<Integer> pila = new piladinamica<>();

    resolver(0, 0, pila); // usando la pila dinamica
    pila = null;
}

```

## Modularidad

La modularidad consiste en dividir un programa en módulos o trozos de códigos que puedan compilarse por separado, sin embargo tendrá conexiones con otros módulos, de esta manera puede organizarse mejor y reducir la complejidad de las clases y funciones.



## Herencia

Es cuando una clase nueva se crea a partir de una clase existente. La herencia (a la que habitualmente se denomina **subclase**) proviene del hecho de que la subclase (la nueva clase creada) contiene los atributos y métodos de la clase primaria.

```
public class Algoritmo implements Acciones{
```

## Ligadura dinámica de métodos

La ligadura dinámica se encarga de ligar o relacionar la llamada a un método con el cuerpo del método que se ejecuta finalmente dependiendo dinámicamente de la implementación del código de la clase a la cual se invoca.

Se ejecuta la implementación definida en la clase Programa:

```
Acciones p = new Programa();  
p.comenzar();
```

## Backtracking

**Backtracking** (o búsqueda atrás) es una técnica de **programación** para hacer búsqueda sistemática a través de todas las configuraciones posibles dentro de un espacio de búsqueda. Para lograr esto, los algoritmos de tipo **backtracking** construyen posibles soluciones candidatas de manera sistemática

Se elimina la última reina y se pasa a la siguiente columna para buscar más soluciones.

```
columnaActual = pila.pop() + 1;  
  
resolver(columnaActual, solucionesEncontradas, pila);
```

## Pila

Es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, siendo el modo de acceso a sus elementos de tipo LIFO (del inglés *Last In, First Out*, «último en entrar, primero en salir»)

```
piladinamica<Integer> pila = new piladinamica<>();
```

## Conclusión

El algoritmo de las 8 reinas es un caso excepcional para aplicar técnicas de búsquedas como backtracking o fuerza bruta. Sin embargo ambas difieren en términos de optimización y rapidez en cuanto a buscar las soluciones.

El algoritmo puede crearse recursivamente, de una forma más elegante y entendible o a través de ciclos, como en pocas ocasiones se suele emplear.

Una técnica muy importante para la resolución de problemas y implementaciones para la creación de inteligencias artificiales las cuales podrían evolucionar la vida cotidiana de las personas.