



Sesión 3: Pruebas de caja blanca

Diseño de pruebas de caja blanca

Método de prueba de caminos

PPSS Pruebas de caja blanca (*White-box testing*)

- ➔ Las pruebas de caja blanca (al igual que las de caja negra) son DINÁMICAS:
 - ⦿ Necesitamos ejecutar el código para detectar errores
- ➔ Las pruebas de caja blanca también se denominan pruebas ESTRUCTURALES
- ➔ Los casos de prueba se obtienen a partir del **CÓDIGO** de la unidad a probar, siendo también necesario conocer la especificación de lo que debería hacer dicho código
 - ⦿ Los métodos de diseño de pruebas de caja blanca suelen aplicarse solamente a las **UNIDADES**
- ➔ Existen múltiples métodos de caja blanca:
 - ⦿ **Método de caminos** (*basis path testing*)
 - ⦿ Método de flujo de control (*control-flow testing*)
 - ▶ Cobertura de métodos, sentencias, ramas, condiciones
 - ⦿ Método de flujo de datos (*data flow testing*)

PPSS Pruebas de caminos (*Basis path testing*)

- ➔ Es un método de diseño de pruebas de caja blanca cuyo **OBJETIVO** es ejercitar (ejecutar) cada **camino independiente** en el programa
 - ⦿ Fue propuesto inicialmente por Tom McCabe en 1976. Este método permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esta medida como guía para la definición de un conjunto básico de caminos de ejecución
- ➔ Si ejecutamos TODOS los caminos independientes, estaremos ejecutando **TODAS** las **sentencias** del programa, al menos una vez
 - ⦿ Además estaremos garantizando que **TODAS** las **condiciones** se ejecutan en sus casos verdadero/falso
- ➔ ¿Qué es un camino independiente?
 - ⦿ Es un camino en un grafo de flujo que representa un “camino” del programa que difiere de otros caminos en al menos un nuevo conjunto de sentencias y/o una nueva condición (Pressman, 2001)

PPSS Pasos para diseñar las pruebas

1. Construir el grafo de flujo del programa a partir del código a probar
2. Calcular la complejidad ciclomática del grafo de flujo
3. Obtener los caminos independientes del grafo
4. Determinar los datos de prueba de entrada de forma que se ejerciten todos los caminos independientes, para cada camino
5. Determinar el resultado esperado para cada camino, en función de la especificación de la unidad a probar

Resultado del diseño de las pruebas de caminos:

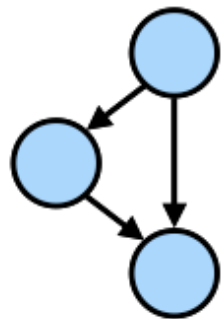
Camino	Datos Entrada	Resultado Esperado	Resultado Real
C1	d1, d2, ...dn	r1	
C2	d1, d2, ...dn	r2	

Recuerda que...

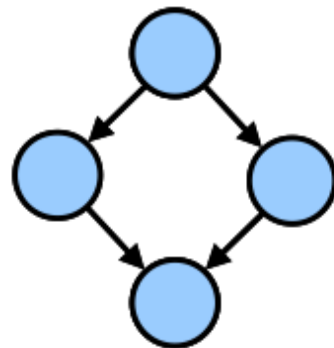
Los datos de entrada y el resultado esperado deben ser CONCRETOS!!!

PPSS Paso 1: Grafo de flujo (I)

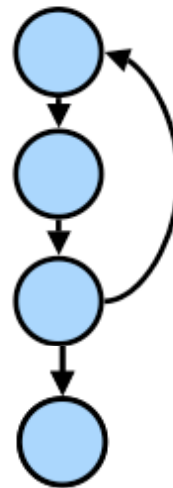
- ➔ Representa el flujo de ejecución de la unidad a probar
- ➔ Los nodos representan sentencias del programa
- ➔ Los arcos representan el flujo de control
- ➔ Cada nodo **SOLAMENTE** puede representar una **UNICA condición** (un nodo puede contener cualquier número de sentencias secuenciales)
- ➔ Ejemplos:



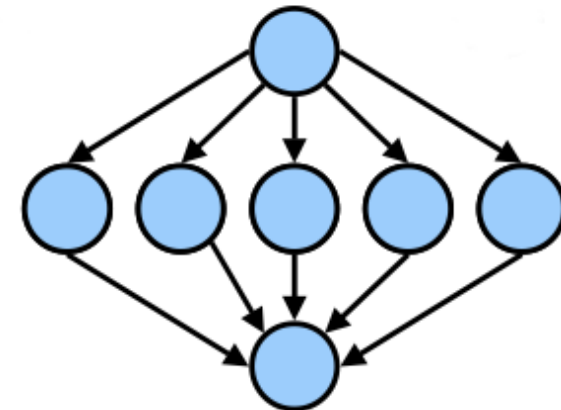
if-then



if-then-else



loop (do-while)

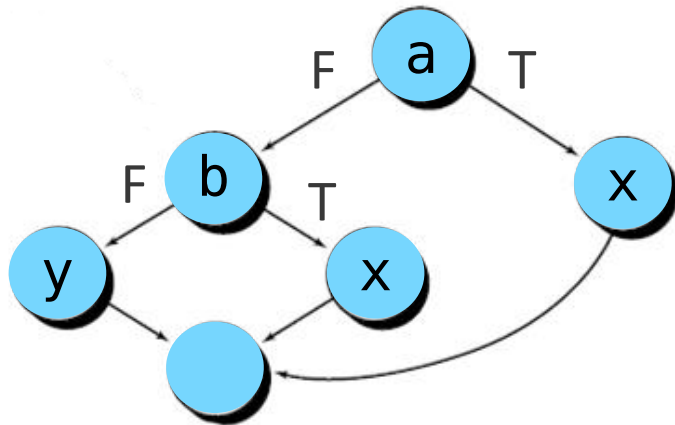


case statement

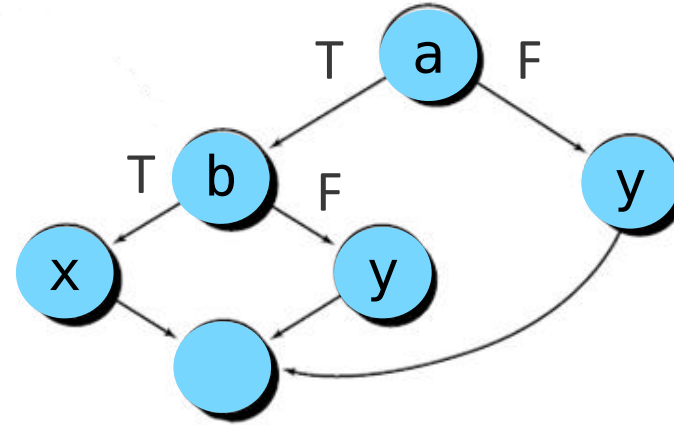
PPSS Paso 1: Grafo de flujo (II)

→ Ejemplos de grafos de flujo asociados a condiciones compuestas

• if (a OR b) then {x} else {y}

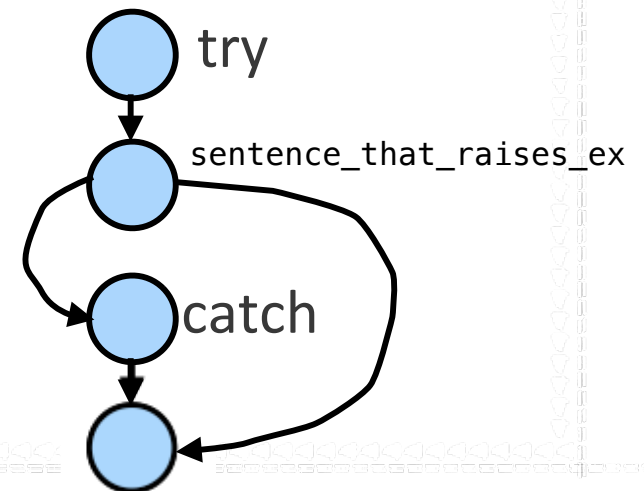


• if (a AND b) then {x} else {y}



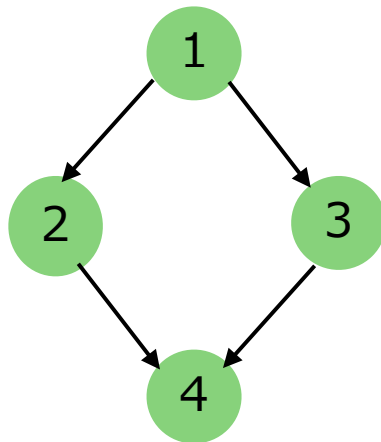
→ Grafo de flujo asociado a una sentencia *try...catch*

```
try {  
    sentence_that_raises_ex;  
} catch (Exception ex) {  
    ...  
}
```



PPSS Paso 2: Complejidad ciclomática (I)

- ➔ Es una métrica que proporciona una medida de la complejidad lógica de un componente software
- ➔ Se calcula a partir del grafo de flujo:
 - ⦿ $CC = \text{número de arcos} - \text{número de nodos} + 2$
- ➔ El valor de CC indica el MÁXIMO número de caminos independientes en el grafo
- ➔ Ejemplo:



$$CC = 4 - 4 + 2 = 2$$

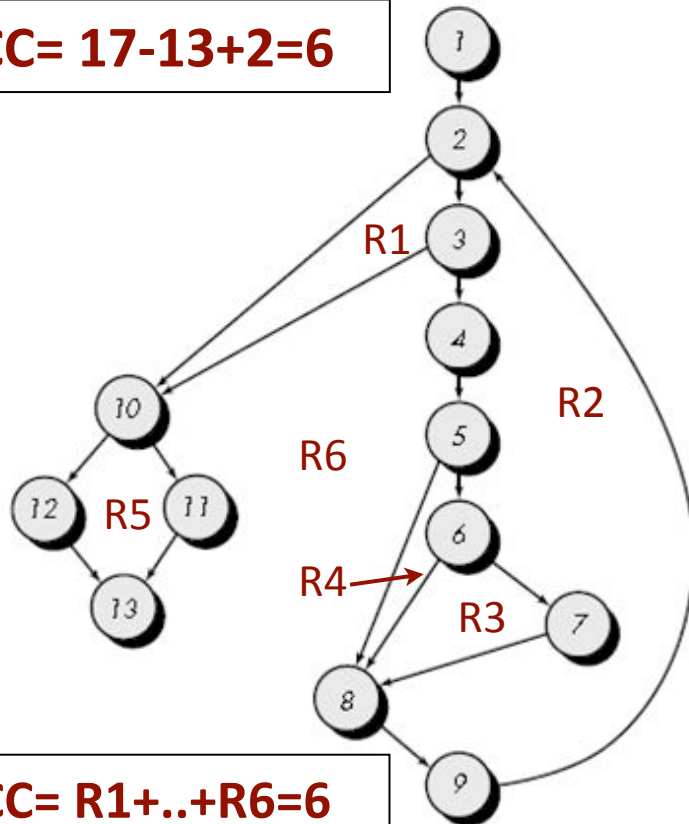
A mayor CC, mayor complejidad lógica, por lo tanto, mayor esfuerzo de mantenimiento, y también requiere un mayor esfuerzo de pruebas!!!

PPSS Paso 2: Complejidad ciclomática (II)

→ Formas alternativas de obtener la complejidad ciclomática:

- $CC = \text{número de arcos} - \text{número de nodos} + 2$
- $CC = \text{número de regiones}$
- $CC = \text{número de condiciones} + 1$

$$CC = 17 - 13 + 2 = 6$$



$$CC = R1 + \dots + R6 = 6$$

¡CUIDADO!: Las dos últimas formas de cálculo son aplicables SOLO si el código es totalmente estructurado (no saltos incondicionales)

```
...  
i=1; total.input=total.valid=0; sum=0;  
do while value[i] <> -999 AND total.input < 100  
  total.input+=1; 2 5 3  
  if value[i] >= minimum and value[i] <= maximum 6  
  then {total.valid+=1;  
        7 sum= sum + value[i];}  
  i+=1; 9  
}  
if total.valid > 0 average= sum/total.valid 10  
else average = -999;  
return average; 13
```

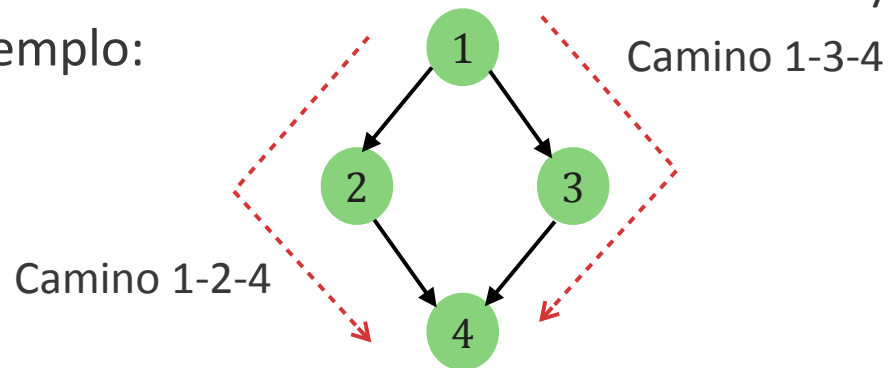
$$CC = 5 + 1 = 6$$

PPSS Paso 3: Caminos independientes

- ➔ Buscamos (como máximo) tantos caminos independientes como valor obtenido de CC

- Cada camino independiente contiene un nodo, o bien una arista, que no aparece en el resto de caminos independientes
- Con ellos recorreremos TODOS los nodos y TODAS las aristas del grafo

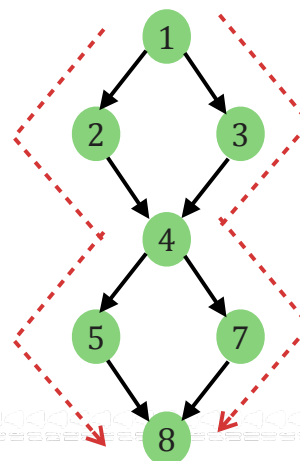
- ➔ Ejemplo:



- ➔ Es posible que con un número inferior a CC recorramos todos los nodos y todas las aristas

- Ejemplo:

```
if (a>=20) {result=0;
} else {
  result=10;
}
if (b>=20) {result=0;
} else {
  result=10;
}
```



CC=3

opción 1:

C1 = 1-3-4-7-8
C2 = 1-3-4-5-8
C3 = 1-2-4-5-8

opción 2:

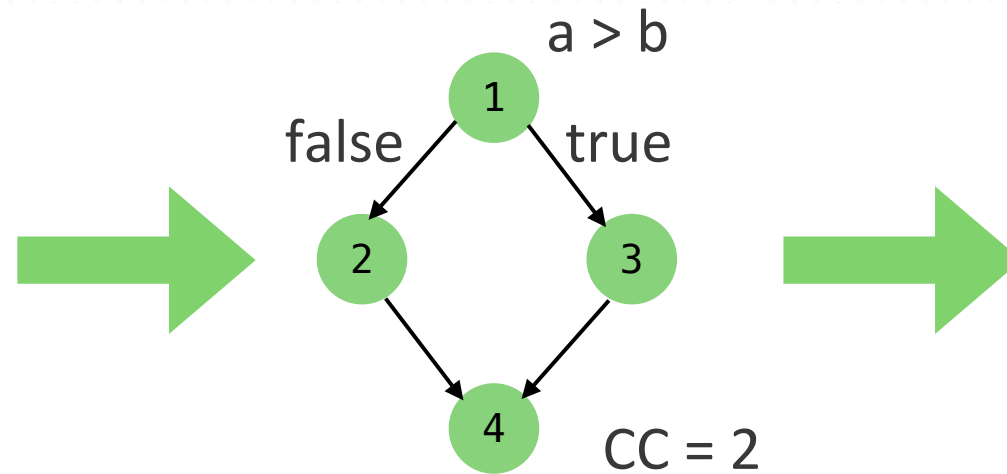
C1 = 1-3-4-7-8
C2 = 1-2-4-5-8

Ambas opciones son válidas

PPSS Paso 4: Datos de entrada para cada camino independiente

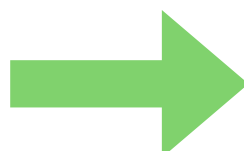
→ Ejemplo:

```
if (a > b) {  
  result = 20  
} else {  
  result = 0  
}
```



C1 = 1-3-4
C2 = 1-2-4

Tabla resultante del diseño de casos de prueba



Camino	Datos Entrada	Resultado Esperado	Resultado Real
C1	a = 20; b = 10	result = 20	
C2	a = 10; b = 20	result = 0	

PPSS Ejemplo: Búsqueda binaria

//Asumimos que la lista de elementos está ordenada de forma ascendente

```
class BinSearch
{
    public static void search (int key, int [ ] elemArray, Result r)
    {
        int bottom = 0;    int top = elemArray.length -1;
        int mid;           r.found= false; r.index= -1;
        while (bottom <= top) {
            mid = (top+bottom)/2;
            if (elemArray [mid] == key) {
                r.index = mid;
                r.found = true;
                return;
            } else {
                if (elemArray [mid] < key)
                    bottom = mid + 1;
                else top = mid -1;
            }
        } //while loop
    } //search
} //class
```

PPSS Vamos a identificar los nodos del grafo

//Asumimos que la lista de elementos está ordenada de forma ascendente

```
class BinSearch
```

```
public static void search (int key, int [ ] elemArray, Result r)
```

```
{   int bottom = 0;   int top = elemArray.length - 1;  
    int mid;           r.found= false; r.index= -1;
```

```
    while (bottom <= top) {
```

```
        mid = (top+bottom)/2;
```

```
        if (elemArray [mid] == key) {
```

```
            r.index = mid;
```

```
            r.found = true;
```

```
            return;
```

```
        } else {
```

```
            if (elemArray [mid] < key)
```

```
                bottom = mid + 1;
```

```
            else top = mid - 1;
```

```
        }
```

```
    } //while loop
```

```
} //search
```

```
} //class
```

1

2

3

8

4

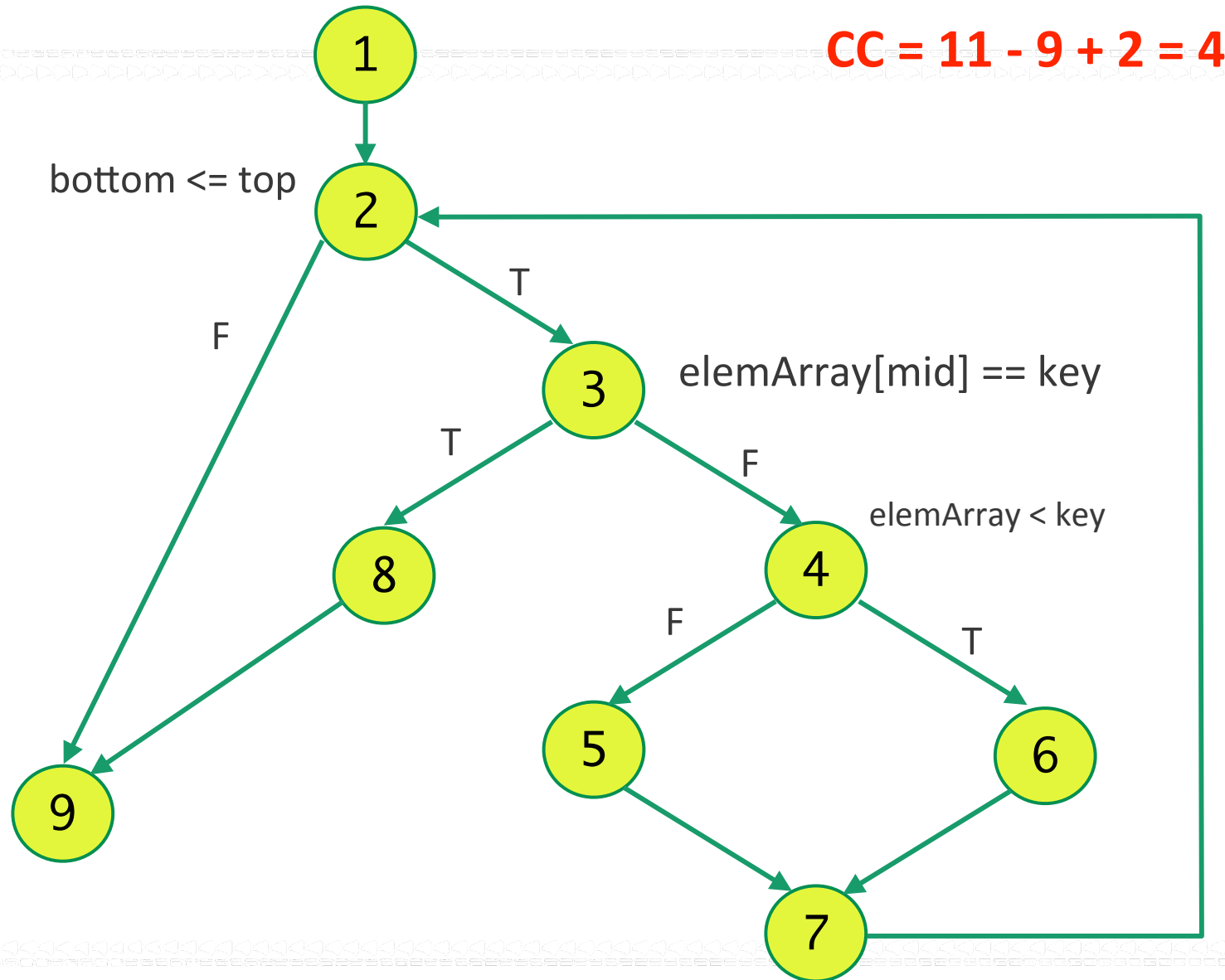
5

6

7

9

PPSS Grafo asociado y valor de CC



PPSS Caminos independientes

- C1: 1, 2, 3, 4, 6, 7, 2, 9
- C2: 1, 2, 3, 4, 5, 7, 2, 9
- C3: 1, 2, 3, 8, 9

En este ejemplo, con tres caminos podemos recorrer todos los nodos y todas las aristas

➔ Ejercicio: Calcula la tabla resultante:

Camino	Datos Entrada	Resultado Esperado	Resultado Real
C1	key= elemArray=	r.found= r.index=	
C2	key= elemArray=	r.found= r.index=	
C3	key= elemArray=	r.found= r.index=	

PPSS Ejercicio propuesto 1

→ Calcula la CC para cada uno de estos códigos Java:

```
public void divide(int numberToDivide, int numberToDivideBy) throws BadNumberException{
    if(numberToDivideBy == 0){
        throw new BadNumberException("Cannot divide by 0");
    }
    return numberToDivide / numberToDivideBy;
}
```

Código 1

```
public void callDivide(){
    try {
        int result = divide(2,1);
        System.out.println(result);
    } catch (BadNumberException e) {
        //do something clever with the exception
        System.out.println(e.getMessage());
    }
    System.out.println("Division attempt done");
}
```

Código 2

```
public void openFile(){
    try {
        // constructor may throw FileNotFoundException
        FileReader reader = new FileReader("someFile");
        int i=0;
        while(i != -1){
            //reader.read() may throw IOException
            i = reader.read();
            System.out.println((char) i );
        }
        reader.close();
        System.out.println("---- File End ----");
    } catch (FileNotFoundException e) {
        //do something clever with the exception
    } catch (IOException e) {
        //do something clever with the exception
    }
}
```

Código 3

PPSS Ejercicio propuesto 2

- ➔ Diseñar los casos de prueba para el método `validar_PIN()`, cuyo código es el siguiente:

```
1.  public class Cajero {
2.  ...
3.  public boolean validar_PIN (Pin pin_number) {
4.      boolean pin_valido= false;
5.      String codigo_respuesta="GOOD";
6.      int contador_pin= 0;
7.
8.      while ((!pin_valido) && (contador_pin <= 2) &&
9.              !codigo_respuesta.equals("CANCEL")) {
10.         codigo_respuesta = obtener_pin(pin_number);
11.         if (!codigo_respuesta.equals("CANCEL")) {
12.             pin_valido = comprobar_pin(pin_number);
13.             if (!pin_valido) {
14.                 System.out.println("PIN inválido, repita");
15.                 contador_pin=contador_pin+1;
16.             }
17.         }
18.     }
19.     return pin_valido;
20. }
21. ...
22. }
```


PPSS Ejercicio propuesto 2

- El método `validar_PIN()` valida un código numérico de cuatro cifras (objeto de la clase Pin) introducido a través de un teclado (asumimos que en el teclado solamente hay teclas numéricas (0..9), y una tecla para cancelar). El método `obtener_pin()` “lee” el código introducido por teclado creando una nueva instancia de un objeto Pin, y devuelve “GOOD” si no se pulsa la tecla para cancelar, o “CANCEL” si se ha pulsado la tecla para cancelar (carácter ‘\’). El método `comprobar_pin()` verifica que el código introducido tiene cuatro cifras y se corresponde con la contraseña almacenada en el sistema para dicho usuario, devolviendo cierto o falso, en función de ello. El usuario dispone de tres intentos para introducir un pin válido, en cuyo caso el método `validar_PIN()` devuelve cierto, y en caso contrario devuelve falso.

PPSS Referencias

- ➔ Software engineering, 9th ed. Ian Sommerville. Addison Wesley. 2011
 - Capítulo 8: Software Testing
- ➔ A practitioner's guide to software test design. Lee Copeland. Artech House Publishers. 2007
 - Capítulo 10: Control Flow Testing
- ➔ Pragmatic software testing. Rex Black. Wiley. 2007
 - Capítulo 21: Control Flow Testing