

Ejercicios resueltos de Entornos de Desarrollo

Los 19 ejercicios de Entornos de Desarrollo de la web, tema a tema: cada uno con su enunciado, los datos que necesitas (código de partida, ejemplos de entrada y salida o la base de datos) y, al final, la solución explicada.

Índice

Desarrollo de Software

[1. Gráfico de trabajo pendiente \(burndown\) de un sprint](#)

[2. Planificación con dependencias y camino crítico](#)

Instalación y uso de IDEs

[3. Depura el resumen de notas](#)

[4. Depura una búsqueda binaria](#)

Diseño y realización de pruebas

[5. Valores límite: casos que detectan errores](#)

[6. Matar mutantes con varias condiciones](#)

Optimización y Documentación

[7. Un analizador de código en miniatura](#)

[8. Un Git en miniatura: base de fusión](#)

Pruebas con JUnit y Mockito

[9. Dobles de prueba escritos a mano](#)

[10. Tu propio JUnit con reflexión](#)

Diagramas UML

[11. Del diagrama de estados al código](#)

[12. Ingeniería inversa: de las clases al diagrama](#)

Ejercicios de Entornos de Desarrollo

[13. Depura el programa de notas](#)

[14. Valores límite: clasificar edades](#)

[15. Caja negra: el problema del triángulo](#)

[16. Refactoriza: extraer método](#)

[17. Del diagrama de clases al código](#)

Ejercicios largos

[18. Validador de DNI, NIE e IBAN diseñado con clases de equivalencia](#)

[19. Depura el programa de notas: cinco errores escondidos](#)

Desarrollo de Software

1. Gráfico de trabajo pendiente (burndown) de un sprint

Difícil · Desarrollo de Software · apuntesdam.com/subject/entornos/topic/desarrollo-software

En Scrum, el burndown muestra cada día cuántos puntos de historia quedan por terminar. La entrada es la duración del sprint en días y, después, cada historia con su identificador, sus puntos y el día en que se terminó (0 si no se terminó). Muestra la cabecera «Sprint de D días, T puntos comprometidos» y, para cada día de 0 a D, una línea con el formato «Día %2d | barra | %2d (ideal %.1f)», donde la barra son tantos # como puntos quedan, rellena con espacios hasta T caracteres, y el ideal es la recta que baja de T a 0 ($T - T \cdot d/D$). Si quedan más puntos que el ideal redondeado hacia arriba, añade « (retraso)». Una historia solo descuenta sus puntos al terminarse (no hay puntos a medias). Al final, la velocidad (puntos terminados) y las historias que vuelven al product backlog (o «ninguna»).

Código de partida (java)

```
import java.util.*;

public class Main {
    record Historia(String id, int puntos, int diaTerminada) { }    // diaTerminada = 0 si no se terminó

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int dias = sc.nextInt();
        List<Historia> hs = new ArrayList<>();
        while (sc.hasNext()) hs.add(new Historia(sc.next(), sc.nextInt(), sc.nextInt()));
        // TODO: mostrar el burndown día a día, la velocidad y las historias pendientes
    }
}
```

Ejemplo: Sprint que acaba bien

ENTRADA

```
5
H1 3 2
H2 5 4
H3 2 5
```

SALIDA ESPERADA

```
Sprint de 5 días, 10 puntos comprometidos
Día 0 | ##### | 10 (ideal 10.0)
Día 1 | ##### | 10 (ideal 8.0) (retraso)
Día 2 | ##### | 7 (ideal 6.0) (retraso)
Día 3 | ##### | 7 (ideal 4.0) (retraso)
Día 4 | ## | 2 (ideal 2.0)
Día 5 | | 0 (ideal 0.0)
Velocidad: 10 puntos
Vuelven al product backlog: ninguna
```

Ejemplo: Sprint con retraso

ENTRADA

```
4
H1 5 3
H2 3 0
H3 2 4
H4 1 1
```

SALIDA ESPERADA

```
Sprint de 4 días, 11 puntos comprometidos
Día 0 | ##### | 11 (ideal 11.0)
Día 1 | ##### | 10 (ideal 8.3) (retraso)
Día 2 | ##### | 10 (ideal 5.5) (retraso)
Día 3 | ##### | 5 (ideal 2.8) (retraso)
Día 4 | ### | 3 (ideal 0.0) (retraso)
Velocidad: 8 puntos
Vuelven al product backlog: H2
```

2. Planificación con dependencias y camino crítico

Muy difícil · Desarrollo de Software · apuntesdam.com/subject/entornos/topic/desarrollo-software

Para planificar un proyecto se descompone en tareas con su duración y las tareas que deben terminar antes (como en un diagrama de Gantt o PERT). Cada línea de la entrada es «ID duración» y, opcionalmente, la lista de tareas previas separadas por comas. Si una tarea depende de otra que no existe, muestra «Error: X depende de Y, que no existe».

Calcula un orden topológico con el algoritmo de Kahn: empieza por las tareas sin previas en el orden de la entrada y usa una cola; cada tarea, al salir, libera a las que dependen de ella (en el orden en que se declararon). Si no se pueden ordenar todas, «Error: hay dependencias circulares». Después calcula el inicio más temprano (el máximo fin de sus previas) y el fin de cada tarea, la duración del proyecto, el fin más tardío de cada tarea sin retrasar el proyecto (recorriendo el orden al revés) y la holgura (fin tardío – fin temprano). Muestra el orden, una línea por tarea con «%-3s inicio %2d fin %2d holgura %d», la duración y el camino crítico: las tareas con holgura 0, en orden topológico, unidas por « → ».

Código de partida (java)

```
import java.util.*;

public class Main {
    record Tarea(String id, int duracion, List<String> previas) { }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, Tarea> tareas = new LinkedHashMap<>();
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.isEmpty()) continue;
            // Formato: ID duración [previa1,previa2,...]
            String[] p = l.split("\\s+");
            List<String> previas = p.length > 2 ? List.of(p[2].split(",")) : List.of();
            tareas.put(p[0], new Tarea(p[0], Integer.parseInt(p[1]), previas));
        }
        // TODO: orden topológico, inicio y fin más tempranos, duración del proyecto,
        //         holgura de cada tarea y camino crítico (o error si hay un ciclo)
    }
}
```

Ejemplo: Proyecto web

ENTRADA

```
A 3
B 2 A
C 4 A
D 1 B,C
E 2 D
```

SALIDA ESPERADA

```
Orden: A B C D E
A inicio 0 fin 3 holgura 0
B inicio 3 fin 5 holgura 2
C inicio 3 fin 7 holgura 0
D inicio 7 fin 8 holgura 0
E inicio 8 fin 10 holgura 0
Duración del proyecto: 10 días
Camino crítico: A → C → D → E
```

Ejemplo: Dos ramas

ENTRADA

```
Analisis 5
BD 3 Analisis
API 6 BD
UI 4 Analisis
Pruebas 2 API,UI
Manual 1 UI
```

SALIDA ESPERADA

```
Orden: Analisis BD UI API Manual Pruebas
Analisis inicio 0 fin 5 holgura 0
BD inicio 5 fin 8 holgura 0
UI inicio 5 fin 9 holgura 5
API inicio 8 fin 14 holgura 0
Manual inicio 9 fin 10 holgura 6
Pruebas inicio 14 fin 16 holgura 0
Duración del proyecto: 16 días
Camino crítico: Analisis → BD → API → Pruebas
```

Instalación y uso de IDEs

3. Depura el resumen de notas

Difícil · Instalación y uso de IDEs · apuntesdam.com/subject/entornos/topic/ides

El método resumen debería calcular la media (con decimales), cuántas notas son aprobados (5 o más), la nota máxima y la mínima de un grupo de notas enteras de 0 a 10. Compila y parece funcionar, pero tiene cuatro errores. Usa el depurador de tu IDE (pon un punto de ruptura dentro del bucle y observa i, suma, max y min en cada vuelta) o añade trazas temporales, encuentra los cuatro y corrígelos sin reescribir el método desde cero.

Código de partida (java)

```
import java.util.*;

public class Main {
    // Este código compila pero tiene CUATRO errores. Encuéntralos con el depurador (o con trazas) y corrígelos.
    static String resumen(int[] notas) {
        int suma = 0, aprobados = 0, max = 0, min = 0;
        for (int i = 1; i < notas.length; i++) {
            suma += notas[i];
            if (notas[i] > 5) aprobados++;
            if (notas[i] > max) max = notas[i];
            if (notas[i] < min) min = notas[i];
        }
        double media = suma / notas.length;
        return String.format(Locale.US, "media %.2f, aprobados %d de %d, máxima %d, mínima %d",
            media, aprobados, notas.length, max, min);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.isEmpty()) continue;
            int[] notas = Arrays.stream(l.split("\\s+")).mapToInt(Integer::parseInt).toArray();
            System.out.println(Arrays.toString(notas) + " → " + resumen(notas));
        }
    }
}
```

Ejemplo: Grupo variado

ENTRADA

7 4 9 5 10

SALIDA ESPERADA

[7, 4, 9, 5, 10] → media 7.00, aprobados 4 de 5, máxima 10, mínima 4

Ejemplo: Todos suspensos

ENTRADA

3 1 4

SALIDA ESPERADA

[3, 1, 4] → media 2.67, aprobados 0 de 3, máxima 4, mínima 1

4. Depura una búsqueda binaria

Muy difícil · Instalación y uso de IDEs · apuntesdam.com/subject/entornos/topic/ides

primeraMayorIgual debe devolver la primera posición de un array ordenado cuyo valor es mayor o igual que x, o a.length si todos son menores (es el «lower bound» que usan Arrays.binarySearch y las bases de datos en sus índices). Con ella, contar calcula cuántas veces aparece x. La versión actual tiene varios errores: según el caso, se queda en un

bucle infinito, devuelve una posición equivocada o nunca devuelve a.length. Depúrala con casos pequeños (arrays de 1 o 2 elementos), anotando lo, hi y medio en cada vuelta, y corrígela. Además, el cálculo de medio puede desbordarse con arrays de más de mil millones de elementos: cámbialo por una forma segura.

Código de partida (java)

```
import java.util.*;

public class Main {
    // Primera posición i tal que a[i] >= x (a.length si no hay ninguna). El array está ordenado.
    // Tiene errores que solo aparecen en algunos casos: encuéntralos.
    static int primeraMayorOIgual(int[] a, int x) {
        int lo = 0, hi = a.length - 1;
        int vueltas = 0; // protección para que no se cuelgue mientras lo
        depuras
        while (lo < hi && vueltas++ < 100) {
            int medio = (lo + hi) / 2;
            if (a[medio] < x) lo = medio;
            else hi = medio - 1;
        }
        return lo;
    }

    static int contar(int[] a, int x) {
        return primeraMayorOIgual(a, x + 1) - primeraMayorOIgual(a, x);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[] a =
            Arrays.stream(sc.nextLine().trim().split("\\s+")).mapToInt(Integer::parseInt).toArray();
        while (sc.hasNextInt()) {
            int x = sc.nextInt();
            System.out.println("x=" + x + ": primera posición >= x → " + primeraMayorOIgual(a, x) + ",
                apariciones → " + contar(a, x));
        }
    }
}
```

Ejemplo: Con repetidos

ENTRADA

```
1 3 3 3 5 8
3
4
0
9
```

SALIDA ESPERADA

```
x=3: primera posición >= x → 1, apariciones → 3
x=4: primera posición >= x → 4, apariciones → 0
x=0: primera posición >= x → 0, apariciones → 0
x=9: primera posición >= x → 6, apariciones → 0
```

Ejemplo: Arrays pequeños

ENTRADA

```
7
7
8
6
```

SALIDA ESPERADA

```
x=7: primera posición >= x → 0, apariciones → 1
x=8: primera posición >= x → 1, apariciones → 0
x=6: primera posición >= x → 0, apariciones → 0
```

5. Valores límite: casos que detectan errores

Difícil · Diseño y realización de pruebas · apuntesdam.com/subject/entornos/topic/disenio-pruebas

La técnica de valores límite dice que los errores se esconden en las fronteras entre clases de equivalencia. El método tarifa ya está bien hecho; tu trabajo es escribir en CASOS las pruebas, cada una con un peso y la tarifa esperada. El programa comprueba que tus casos pasan con el método correcto y después los ejecuta contra seis versiones con un error cada una (mutantes), como los que comete cualquier programador: un $<$ en vez de $\leq$, un precio equivocado o un límite olvidado. Un mutante queda «detectado» si alguno de tus casos da un resultado distinto. Consigue detectar los seis con casos razonados, no al azar.

Código de partida (java)

```
import java.util.*;
import java.util.function.DoubleUnaryOperator;

public class Main {
    // Tarifa de envío según el peso en kg. -1 significa «no admitido».
    // peso <= 0      → -1
    // 0 < peso <= 1  → 4.5
    // 1 < peso <= 5  → 7
    // 5 < peso <= 20 → 12
    // peso > 20     → -1
    static double tarifa(double peso) {
        if (peso <= 0) return -1;
        if (peso <= 1) return 4.5;
        if (peso <= 5) return 7;
        if (peso <= 20) return 12;
        return -1;
    }

    // TODO: escribe aquí tus casos de prueba: {peso, resultado esperado}
    static final double[][] CASOS = {
        {3, 7},
    };

    // ---- No modifiques lo que sigue: versiones con un error cada una («mutantes»)
    static final Map<String, DoubleUnaryOperator> MUTANTES = new LinkedHashMap<>();
    static {
        MUTANTES.put("M1 (< 1 en vez de <= 1)", p -> p <= 0 ? -1 : p < 1 ? 4.5 : p <= 5 ? 7 : p <= 20 ?
            12 : -1);
        MUTANTES.put("M2 (< 5 en vez de <= 5)", p -> p <= 0 ? -1 : p <= 1 ? 4.5 : p < 5 ? 7 : p <= 20 ?
            12 : -1);
        MUTANTES.put("M3 (< 20 en vez de <= 20)", p -> p <= 0 ? -1 : p <= 1 ? 4.5 : p <= 5 ? 7 : p < 20 ?
            12 : -1);
        MUTANTES.put("M4 (< 0 en vez de <= 0)", p -> p < 0 ? -1 : p <= 1 ? 4.5 : p <= 5 ? 7 : p <= 20 ?
            12 : -1);
        MUTANTES.put("M5 (sin límite superior)", p -> p <= 0 ? -1 : p <= 1 ? 4.5 : p <= 5 ? 7 : 12);
        MUTANTES.put("M6 (7.5 en vez de 7)", p -> p <= 0 ? -1 : p <= 1 ? 4.5 : p <= 5 ? 7.5 : p <= 20 ?
            12 : -1);
    }

    public static void main(String[] args) {
        String fallo = null;
        for (double[] c : CASOS)
            if (tarifa(c[0]) != c[1] && fallo == null) fallo = "tarifa(" + c[0] + ") da " + tarifa(c[0])
                + ", no " + c[1];
        System.out.println("Casos correctos con el original: " + (fallo == null ? "sí" : "no (" + fallo +
            ")"));
        List<String> vivos = new ArrayList<>();
        for (var m : MUTANTES.entrySet()) {
            boolean detectado = false;
            for (double[] c : CASOS) if (m.getValue().applyAsDouble(c[0]) != c[1]) detectado = true;
            if (!detectado) vivos.add(m.getKey());
        }
        System.out.println("Mutantes detectados: " + (MUTANTES.size() - vivos.size()) + " de " +
            MUTANTES.size());
        System.out.println("Sobreviven: " + (vivos.isEmpty() ? "ninguno" : String.join(", ", vivos)));
    }
}
```

Ejemplo: Detección de mutantes

SALIDA ESPERADA

```
Casos correctos con el original: sí
Mutantes detectados: 6 de 6
Sobreviven: ninguno
```

6. Matar mutantes con varias condiciones

Muy difícil · Diseño y realización de pruebas · apuntesdam.com/subject/entornos/topic/disenio-pruebas

El precio de la entrada de un museo depende de la edad, de si se es socio y de si es fin de semana (las reglas están en el comentario de ORIGINAL). Escribe en CASOS las pruebas necesarias, cada una con la edad, si es socio (1/0), si es fin de semana (1/0) y el precio esperado, para detectar los mutantes: errores en los límites de edad, en el orden en que se aplican los descuentos, en el tope de 0 € o en cómo se combinan las condiciones. Atención: uno de los ocho es un mutante equivalente, que se comporta exactamente igual que el original con cualquier entrada posible, así que ningún caso puede detectarlo. Detecta los otros siete y averigua cuál es el equivalente y por qué. Con muchas condiciones, probar todas las combinaciones crece muy deprisa: busca un conjunto pequeño (con unos 12 casos basta) en el que cada condición y cada límite influyan en algún resultado.

```

import java.util.*;

public class Main {
    interface Precio { double calcular(int edad, boolean socio, boolean finDeSemana); }

    // Precio de la entrada a un museo:
    //  edad < 0 → -1 (error)
    //  base 10 €; los fines de semana, 12 €
    //  menores de 12: 50 % de descuento; desde 65: 40 % de descuento
    //  los socios tienen además 2 € menos sobre el precio ya descontado (nunca por debajo de 0)
    static final Precio ORIGINAL = (edad, socio, finde) -> {
        if (edad < 0) return -1;
        double p = finde ? 12 : 10;
        if (edad < 12) p *= 0.5;
        else if (edad >= 65) p *= 0.6;
        if (socio) p = Math.max(0, p - 2);
        return p;
    };

    // TODO: tus casos: {edad, socio (0/1), fin de semana (0/1), precio esperado}
    static final double[][] CASOS = {
        {30, 0, 0, 10},
    };

    // ---- No modifique lo que sigue
    static final Map<String, Precio> MUTANTES = new LinkedHashMap<>();
    static {
        MUTANTES.put("M1 (<= 12)", (e, s, f) -> e < 0 ? -1 : aplicar(f ? 12 : 10, e <= 12, e >= 65, s));
        MUTANTES.put("M2 (> 65)", (e, s, f) -> e < 0 ? -1 : aplicar(f ? 12 : 10, e < 12, e > 65, s));
        MUTANTES.put("M3 (<= 0 es error)", (e, s, f) -> e <= 0 ? -1 : aplicar(f ? 12 : 10, e < 12, e >= 65, s));
        MUTANTES.put("M4 (socio antes del %)", (e, s, f) -> {
            if (e < 0) return -1;
            double p = (f ? 12 : 10) - (s ? 2 : 0);
            return e < 12 ? p * 0.5 : e >= 65 ? p * 0.6 : p;
        });
        MUTANTES.put("M5 (sin tope en 0)", (e, s, f) -> {
            if (e < 0) return -1;
            double p = f ? 12 : 10;
            if (e < 12) p *= 0.5; else if (e >= 65) p *= 0.6;
            return s ? p - 2 : p;
        });
        MUTANTES.put("M6 (ignora el fin de semana con descuento)", (e, s, f) -> e < 0 ? -1 : aplicar((f
            && e >= 12 && e < 65) ? 12 : 10, e < 12, e >= 65, s));
        MUTANTES.put("M7 (el socio solo descuenta a adultos)", (e, s, f) -> e < 0 ? -1 : aplicar(f ? 12 :
            10, e < 12, e >= 65, s && e >= 12 && e < 65));
        MUTANTES.put("M8 (65 o más: 50 %)", (e, s, f) -> {
            if (e < 0) return -1;
            double p = f ? 12 : 10;
            if (e < 12 || e >= 65) p *= 0.5;
            if (s) p = Math.max(0, p - 2);
            return p;
        });
    };

    static double aplicar(double p, boolean nino, boolean mayor, boolean socio) {
        if (nino) p *= 0.5; else if (mayor) p *= 0.6;
        if (socio) p = Math.max(0, p - 2);
        return p;
    }

    static boolean igual(double a, double b) { return Math.abs(a - b) < 1e-9; }

    public static void main(String[] args) {
        String fallo = null;
        for (double[] c : CASOS) {
            double r = ORIGINAL.calcular((int) c[0], c[1] == 1, c[2] == 1);
            if (!igual(r, c[3]) && fallo == null) fallo = Arrays.toString(c) + " da " + r;
        }
    }
}

```

```

        System.out.println("Casos correctos con el original: " + (fallo == null ? "sí" : "no (" + fallo +
        ")));
        List<String> vivos = new ArrayList<>();
        for (var m : MUTANTES.entrySet()) {
            boolean detectado = false;
            for (double[] c : CASOS)
                if (!igual(m.getValue().calcular((int) c[0], c[1] == 1, c[2] == 1), c[3])) detectado =
                    true;
            if (!detectado) vivos.add(m.getKey());
        }
        System.out.println("Mutantes detectados: " + (MUTANTES.size() - vivos.size()) + " de " +
        MUTANTES.size());
        System.out.println("Sobreviven: " + (vivos.isEmpty() ? "ninguno" : String.join(", ", vivos)));
    }
}

```

Ejemplo: Detección de mutantes

SALIDA ESPERADA

```

Casos correctos con el original: sí
Mutantes detectados: 7 de 8
Sobreviven: M5 (sin tope en 0)

```

Optimización y Documentación

7. Un analizador de código en miniatura

Difícil · Optimización y Documentación · apuntesdam.com/subject/entornos/topic/optimizacion-documentacion

Herramientas como Checkstyle o PMD leen el código fuente y avisan de los «malos olores». Implementa analizar con cuatro reglas, recorriendo las líneas en orden. Detecta la cabecera de un método con la expresión regular METODO (el grupo 2 es el nombre y el 3, los parámetros). Regla 1: si el nombre no sigue lowerCamelCase (empieza por minúscula y solo tiene letras y cifras), «Línea n: el método «X» no sigue lowerCamelCase». Regla 2: si tiene más de MAX_PARAMETROS parámetros, «Línea n: «X» tiene p parámetros (máximo 3)». Regla 3: en cada línea que no declare una constante (que no contenga «final»), cada número distinto de 0 y 1 que no esté dentro de una cadena ni de un comentario // es «Línea n: número mágico N». Regla 4: un método que, desde su cabecera hasta la llave que lo cierra (cuenta { y }), mide más de MAX_LINEAS_METODO líneas da «Línea inicio: «X» mide L líneas (máximo 8)», y este aviso se añade al encontrar la llave de cierre. El main muestra los avisos y el total.

Código de partida (java)

```
import java.util.*;
import java.util.regex.*;

public class Main {
    static final int MAX_LINEAS_METODO = 8, MAX_PARAMETROS = 3;
    static final Pattern METODO = Pattern.compile(
        "^\\s*(public|private|protected|static|\\s)+[\\w<>\\[\\]]+\\s+(\\w+)\\s*\\(((\\^)*\\))\\s*\\\\"
        {");

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<String> lineas = new ArrayList<>();
        while (sc.hasNextLine()) lineas.add(sc.nextLine());
        List<String> avisos = analizar(lineas);
        avisos.forEach(System.out::println);
        System.out.println(avisos.size() + (avisos.size() == 1 ? " aviso" : " avisos"));
    }

    static List<String> analizar(List<String> lineas) {
        List<String> avisos = new ArrayList<>();
        // TODO: recorrer las líneas y aplicar las cuatro reglas
        return avisos;
    }
}
```

Ejemplo: Nombre y número mágico

ENTRADA

```
public class Factura {
    static final double IVA = 0.21;

    public double total(double base) {
        return base * (1 + IVA);
    }

    public double Descuento(double total, int
    unidades) {
        if (unidades > 10) return total * 0.95;
        return total;
    }
}
```

SALIDA ESPERADA

Línea 8: el método «Descuento» no sigue
lowerCamelCase
Línea 9: número mágico 10
Línea 9: número mágico 0.95
3 avisos

Ejemplo: Método largo con muchos parámetros

ENTRADA

```
public class Pedidos {
    private double calcular(String cliente, int
    unidades, double precio, boolean urgente, String
    pais) {
        double total = unidades * precio;
        if (urgente) {
            total += 15;
        }
        if (pais.equals("ES")) {
            total *= 1.21;
        }
        String msg = "Pedido de 99 unidades";
        return total;
    }

    public static void main(String[] args) {
        System.out.println(new
        Pedidos().calcular("ana", 3, 2.5, true, "ES"));
    }
}
```

SALIDA ESPERADA

Línea 2: «calcular» tiene 5 parámetros (máximo 3)
Línea 5: número mágico 15
Línea 8: número mágico 1.21
Línea 2: «calcular» mide 11 líneas (máximo 8)
Línea 15: número mágico 3
Línea 15: número mágico 2.5
6 avisos

8. Un Git en miniatura: base de fusión

Muy difícil · Optimización y Documentación · apuntesdam.com/subject/entornos/topic/optimizacion-documentacion

En Git, cada commit apunta a sus padres (uno normalmente, dos si es una fusión) y una rama es solo un nombre que apunta a un commit. Para fusionar, Git busca la base de fusión: el antepasado común más reciente de los dos commits. Implementa antepasados (el propio commit y todos los que se alcanzan siguiendo padres) y baseDeFusion: de los antepasados comunes, el que no es antepasado de ningún otro común (si hubiera varios, el primero en orden de creación; si no hay ninguno, null). Con eso, el main decide qué haría git merge: «ya actualizado» si el origen es la base, «fast-forward» si el destino es la base (basta con mover la rama) o «fusión de tres vías» en otro caso.

Código de partida (java)

```
import java.util.*;

public class Main {
    static final Map<String, List<String>> padres = new LinkedHashMap<>(); // commit → sus padres
    static final Map<String, String> ramas = new TreeMap<>(); // rama → commit al que
        apunta

    // Todos los antepasados de un commit, incluido él mismo
    static Set<String> antepasados(String c) {
        // TODO
        return new HashSet<>();
    }

    // Antepasado común «más reciente» (el que no es antepasado de otro antepasado común); null si no hay
    static String baseDeFusion(String a, String b) {
        // TODO
        return null;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] t = sc.nextLine().trim().split("\\s+");
            switch (t[0]) {
                case "COMMIT" -> padres.put(t[1], t.length > 2 ? List.of(t[2].split(",")) : List.of());
                case "RAMA" -> ramas.put(t[1], t[2]);
                case "BASE" -> System.out.println("merge-base " + t[1] + " " + t[2] + " → " +
                    baseDeFusion(ramas.get(t[1]), ramas.get(t[2])));
                case "FUSIONAR" -> { // FUSIONAR destino origen: ¿qué haría git merge origen
                    estando en destino?
                    String d = ramas.get(t[1]), o = ramas.get(t[2]);
                    String base = baseDeFusion(d, o);
                    String r;
                    if (o.equals(base)) r = "ya actualizado (nada que fusionar)";
                    else if (d.equals(base)) r = "fast-forward: " + t[1] + " pasa de " + d + " a " + o;
                    else r = "fusión de tres vías (base " + base + ")": se crea un commit de fusión";
                    System.out.println("git merge " + t[2] + " en " + t[1] + " → " + r);
                }
            }
        }
    }
}
```

Ejemplo: Ramas divergentes

ENTRADA

```
COMMIT c1
COMMIT c2 c1
COMMIT c3 c2
COMMIT c4 c3
COMMIT c5 c2
COMMIT c6 c5
RAMA main c4
RAMA login c6
RAMA hotfix c3
BASE main login
FUSIONAR main login
FUSIONAR main hotfix
FUSIONAR hotfix main
```

SALIDA ESPERADA

```
merge-base main login → c2
git merge login en main → fusión de tres vías
(base c2): se crea un commit de fusión
git merge hotfix en main → ya actualizado (nada
que fusionar)
git merge main en hotfix → fast-forward: hotfix
pasa de c3 a c4
```

Ejemplo: Después de fusionar

ENTRADA

```
COMMIT c1
COMMIT c2 c1
COMMIT c3 c2
COMMIT c4 c3
COMMIT c5 c2
COMMIT c6 c5
RAMA main c4
RAMA login c6
RAMA hotfix c3
COMMIT c7 c4,c6
RAMA main c7
BASE main login
FUSIONAR main login
FUSIONAR login main
```

SALIDA ESPERADA

```
merge-base main login → c6
git merge login en main → ya actualizado (nada
que fusionar)
git merge main en login → fast-forward: login
pasa de c6 a c7
```

Pruebas con JUnit y Mockito

9. Dobles de prueba escritos a mano

Difícil · Pruebas con JUnit y Mockito · apuntesdam.com/subject/entornos/topic/pruebas-junit

Para probar ServicioPedidos sin una base de datos ni un servidor de correo reales se usan dobles de prueba: objetos que implementan las mismas interfaces. Completa InventarioFalso, cuyo constructor recibe el stock que devolverá stock() para cualquier producto y que anota cada llamada a descontar como «producto -unidades» en una lista que devuelve descuentos(), y CorreoEspia, que no envía nada: guarda cada envío como «destinatario | asunto» en una lista que devuelve enviados(). Es lo que hace Mockito automáticamente con when(...).thenReturn(...) y verify(...). El main ya prueba el servicio con distintos casos y muestra lo que han registrado tus dobles.

Código de partida (java)

```
import java.util.*;

interface Inventario {
    int stock(String producto);
    void descontar(String producto, int unidades);
}

interface Correo {
    void enviar(String destinatario, String asunto);
}

// Clase que se quiere probar: no sabe si habla con una base de datos real o con un doble
class ServicioPedidos {
    private final Inventario inventario;
    private final Correo correo;

    ServicioPedidos(Inventario inventario, Correo correo) {
        this.inventario = inventario;
        this.correo = correo;
    }

    boolean pedir(String cliente, String producto, int unidades) {
        if (unidades <= 0) return false;
        if (inventario.stock(producto) < unidades) {
            correo.enviar("almacen@tienda.es", "Sin stock: " + producto);
            return false;
        }
        inventario.descontar(producto, unidades);
        correo.enviar(cliente, "Pedido confirmado: " + unidades + " x " + producto);
        return true;
    }
}

// TODO: un doble de Inventario con un stock fijado por la prueba que anote las llamadas a descontar
class InventarioFalso implements Inventario {
    InventarioFalso(int stockFijo) { }
    public int stock(String producto) { return 0; }
    public void descontar(String producto, int unidades) { }
    List<String> descuentos() { return List.of(); }
}

// TODO: un doble de Correo que guarde cada envío como "destinatario | asunto"
class CorreoEspia implements Correo {
    public void enviar(String destinatario, String asunto) { }
    List<String> enviados() { return List.of(); }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] t = sc.nextLine().trim().split(" "); // stock cliente producto unidades
            InventarioFalso inv = new InventarioFalso(Integer.parseInt(t[0]));
            CorreoEspia correo = new CorreoEspia();
            boolean ok = new ServicioPedidos(inv, correo).pedir(t[1], t[2], Integer.parseInt(t[3]));
            System.out.println("pedir(" + t[1] + ", " + t[2] + ", " + t[3] + ") con stock " + t[0] + " → " + ok);
            System.out.println(" descuentos: " + inv.descuentos());
            System.out.println(" correos: " + correo.enviados());
        }
    }
}
```

Ejemplo: Pedido correcto

ENTRADA

```
10 ana@mail.com teclado 3
```

SALIDA ESPERADA

```
pedir(ana@mail.com, teclado, 3) con stock 10 →  
true  
  descuentos: [teclado -3]  
  correos: [ana@mail.com | Pedido confirmado: 3 x  
teclado]
```

Ejemplo: Sin stock y unidades no válidas

ENTRADA

```
2 luis@mail.com raton 5  
5 eva@mail.com monitor 0
```

SALIDA ESPERADA

```
pedir(luis@mail.com, raton, 5) con stock 2 →  
false  
  descuentos: []  
  correos: [almacen@tienda.es | Sin stock: raton]  
pedir(eva@mail.com, monitor, 0) con stock 5 →  
false  
  descuentos: []  
  correos: []
```

10. Tu propio JUnit con reflexión

Muy difícil · Pruebas con JUnit y Mockito · apuntesdam.com/subject/entornos/topic/pruebas-junit

¿Cómo encuentra JUnit los métodos de prueba? Con reflexión: inspecciona la clase en tiempo de ejecución y busca los métodos con la anotación `@Test`. Implementa ejecutar: localiza los métodos anotados con `@Prueba` y `@Antes` de la clase, ordena las pruebas por nombre y, para cada una, crea una instancia nueva de la clase (así una prueba no afecta a otra), ejecuta los métodos `@Antes` y después la prueba. Si termina bien, «✓ nombre»; si lanza `FalloAsercion`, es un fallo: «✗ nombre: mensaje»; si lanza cualquier otra excepción, es un error: «!! nombre: NombreSimpleDeLaExcepción». Al final, «Pruebas: N, correctas: a, fallidas: b, con error: c». Los métodos sin anotación no se ejecutan.

Código de partida (java)

```
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

// Anotaciones propias, como @Test y @BeforeEach de JUnit
@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.METHOD) @interface Prueba { }
@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.METHOD) @interface Antes { }

class FalloAsercion extends RuntimeException {
    FalloAsercion(String m) { super(m); }
}

class Afirmar {
    static void iguales(Object esperado, Object real) {
        if (!Objects.equals(esperado, real)) throw new FalloAsercion("se esperaba <" + esperado + "> pero fue <" + real + ">");
    }
    static void cierto(boolean c, String mensaje) {
        if (!c) throw new FalloAsercion(mensaje);
    }
}

// ---- Las pruebas de ejemplo (no las modifiques)
class PilaTest {
    private Deque<Integer> pila;

    @Antes void preparar() { pila = new ArrayDeque<>(); pila.push(1); pila.push(2); }

    @Prueba void cimaEsElUltimo() { Afirmar.iguales(2, pila.peek()); }
    @Prueba void sacarReduce() { pila.pop(); Afirmar.iguales(1, pila.size()); }
    @Prueba void cadaPruebaEmpiezaLimpia() { Afirmar.iguales(2, pila.size()); }
    @Prueba void fallaAProposito() { Afirmar.iguales(3, pila.size()); }
    @Prueba void lanzaExcepcion() { new ArrayDeque<Integer>().pop(); }
    void noEsUnaPrueba() { throw new IllegalStateException("no debería ejecutarse"); }
}

public class Main {
    // Ejecuta todos los métodos @Prueba de la clase, en orden alfabético, con una instancia nueva cada vez
    static void ejecutar(Class<?> clase) throws Exception {
        // TODO
    }

    public static void main(String[] args) throws Exception {
        ejecutar(PilaTest.class);
    }
}
```

Ejemplo: Pila de ejemplo

SALIDA ESPERADA

```
✓ cadaPruebaEmpiezaLimpia
✓ cimaEsElUltimo
X fallaAProposito: se esperaba <3> pero fue <2>
!! lanzaExcepcion: NoSuchElementException
✓ sacarReduce
Pruebas: 5, correctas: 3, fallidas: 1, con error: 1
```

11. Del diagrama de estados al código

Difícil · Diagramas UML · apuntesdam.com/subject/entornos/topic/diagramas-uml

El comentario de la clase describe el diagrama de estados de un pedido: sus estados, los eventos que provocan cada transición y las acciones asociadas. Implementa aplicar para que, según el estado actual y el evento, cambie de estado y devuelva el nuevo estado con su acción entre corchetes si la tiene («CANCELADO [devolver el dinero]», «PAGADO [reenviar]»). Un paquete perdido vuelve a PAGADO para reenviarlo, pero a la tercera pérdida el pedido pasa a CANCELADO con devolución. Cualquier evento que no tenga una transición desde el estado actual (incluidos todos los eventos en los estados finales) devuelve «transición no permitida (sigue en ESTADO)» sin cambiar nada.

Código de partida (java)

```
import java.util.*;

public class Main {
    // Diagrama de estados de un pedido (estado inicial: CARRITO)
    // CARRITO --pagar--> PAGADO
    // CARRITO --cancelar--> CANCELADO
    // PAGADO --enviar--> ENVIADO
    // PAGADO --cancelar--> CANCELADO [acción: devolver el dinero]
    // ENVIADO --entregar--> ENTREGADO
    // ENVIADO --perder--> PAGADO [acción: reenviar; como mucho 2 veces, después CANCELADO
    // con devolución]
    // ENTREGADO y CANCELADO son estados finales
    enum Estado { CARRITO, PAGADO, ENVIADO, ENTREGADO, CANCELADO }

    static Estado estado = Estado.CARRITO;
    static int perdidas = 0;

    // Aplica un evento y devuelve el texto que describe lo ocurrido
    static String aplicar(String evento) {
        // TODO
        return "";
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNext()) {
            String ev = sc.next();
            Estado antes = estado;
            String r = aplicar(ev);
            System.out.println(antes + " --" + ev + "--> " + r);
        }
        System.out.println("Estado final: " + estado);
    }
}
```

Ejemplo: Camino feliz

ENTRADA

pagar enviar entregar

SALIDA ESPERADA

CARRITO --pagar--> PAGADO
PAGADO --enviar--> ENVIADO
ENVIADO --entregar--> ENTREGADO
Estado final: ENTREGADO

Ejemplo: Eventos no válidos y cancelación

ENTRADA

enviar pagar cancelar entregar

SALIDA ESPERADA

```
CARRITO --enviar--> transición no permitida
(sigue en CARRITO)
CARRITO --pagar--> PAGADO
PAGADO --cancelar--> CANCELADO [devolver el
dinero]
CANCELADO --entregar--> transición no permitida
(sigue en CANCELADO)
Estado final: CANCELADO
```

12. Ingeniería inversa: de las clases al diagrama

Muy difícil · Diagramas UML · apuntesdam.com/subject/entornos/topic/diagramas-uml

Herramientas como IntelliJ o PlantUML pueden generar el diagrama de clases a partir del código. Hazlo tú con reflexión: para cada clase de la lista, en orden, escribe su bloque de PlantUML («interface X {», «abstract class X {» o «class X {»). Dentro, los atributos ordenados por nombre con su visibilidad (+ public, - private, # protected, ~ paquete) como «-dni : String», y después los métodos declarados en la clase, también por nombre, como «+matricular(Asignatura) : void». Un atributo cuyo tipo es otra clase del modelo no se escribe como atributo sino como asociación «Asignatura --> "1" Profesor : titular»; uno de tipo List<X> es una asociación con multiplicidad * («Alumno --> "*" Matricula : matriculas»). Al final, después de todos los bloques y en el orden en que se descubrieron mientras se recorrían las clases (primero las asociaciones de cada clase y luego su herencia), las relaciones, incluidas la herencia «Persona <|-- Alumno» y la realización de interfaces «Evaluable <|.. Alumno».

Código de partida (java)

```
import java.lang.reflect.*;
import java.util.*;

// ---- Modelo de ejemplo (no lo modifique)
abstract class Persona {
    protected String nombre;
    private String dni;
    public String getNombre() { return nombre; }
}
interface Evaluable { double nota(); }
class Alumno extends Persona implements Evaluable {
    private int curso;
    private List<Matricula> matriculas;
    public double nota() { return 0; }
    public void matricular(Asignatura a) { }
}
class Profesor extends Persona {
    private double salario;
    private List<Asignatura> imparte;
}
class Asignatura {
    private String codigo;
    int horas;
    private Profesor titular;
}
class Matricula {
    private Asignatura asignatura;
    private double nota;
}

public class Main {
    static final List<Class<?>> MODELO = List.of(Persona.class, Evaluable.class, Alumno.class,
        Profesor.class, Asignatura.class, Matricula.class);

    static String visibilidad(int mod) {
        return Modifier.isPublic(mod) ? "+" : Modifier.isPrivate(mod) ? "-" : Modifier.isProtected(mod) ?
            "#" : "~";
    }

    // Genera el diagrama de clases en PlantUML a partir de las clases compiladas
    static String plantUml(List<Class<?>> clases) {
        StringBuilder sb = new StringBuilder("@startuml\n");
        // TODO
        return sb.append("@enduml").toString();
    }

    public static void main(String[] args) {
        System.out.println(plantUml(MODELO));
    }
}
```

Ejemplo: Modelo de un instituto

SALIDA ESPERADA

```
@startuml
abstract class Persona {
    -dni : String
    #nombre : String
    +getNombre() : String
}
interface Evaluable {
    +nota() : double
}
class Alumno {
    -curso : int
    +matricular(Asignatura) : void
    +nota() : double
}
class Profesor {
    -salario : double
}
class Asignatura {
    -codigo : String
    ~horas : int
}
class Matricula {
    -nota : double
}
Alumno --> "*" Matricula : matriculas
Persona <|-- Alumno
Evaluable <|-- Alumno
Profesor --> "*" Asignatura : imparte
Persona <|-- Profesor
Asignatura --> "1" Profesor : titular
Matricula --> "1" Asignatura : asignatura
@enduml
```

Ejercicios de Entornos de Desarrollo

13. Depura el programa de notas

Fácil · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

Este programa lee N notas y debe mostrar su media con dos decimales y la nota máxima, pero tiene tres errores. Usa el depurador (o trazas) para encontrarlos y corrégelos sin reescribir el programa. Entrada: N y después las N notas.

Código de partida (java)

```
import java.util.*;

public class Main {

    // Devuelve la media de las notas (con decimales)
    static double media(int[] notas) {
        int suma = 0;
        for (int i = 1; i <= notas.length - 1; i++) {
            suma += notas[i];
        }
        return suma / notas.length;
    }

    // Devuelve la nota más alta
    static int maxima(int[] notas) {
        int max = notas[0];
        for (int n : notas) {
            if (n < max) max = n;
        }
        return max;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] notas = new int[n];
        for (int i = 0; i < n; i++) notas[i] = sc.nextInt();
        System.out.printf(Locale.US, "Media: %.2f\n", media(notas));
        System.out.println("Máxima: " + maxima(notas));
    }
}
```

Ejemplo: Cuatro notas

ENTRADA

4
5 7 8 10

SALIDA ESPERADA

Media: 7.50
Máxima: 10

Ejemplo: Media con decimales

ENTRADA

3
6 6 7

SALIDA ESPERADA

Media: 6.33
Máxima: 7

14. Valores límite: clasificar edades

Fácil · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

Implementa clasificar(edad) según esta especificación: edad < 0 o > 120 → "Error"; 0–11 → "Niño"; 12–17 → "Adolescente"; 18–64 → "Adulto"; 65–120 → "Jubilado". Los tests usan justo los valores límite de cada clase de equivalencia, que es donde suelen esconderse los errores. Entrada: una edad por línea.

Código de partida (java)

```
import java.util.*;

public class Main {

    static String clasificar(int edad) {
        // TODO: implementa la especificación
        return "";
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) {
            System.out.println(clasificar(sc.nextInt()));
        }
    }
}
```

Ejemplo: Límites inferiores

ENTRADA

-1
0
11
12

SALIDA ESPERADA

Error
Niño
Niño
Adolescente

Ejemplo: Límites centrales

ENTRADA

17
18
64
65

SALIDA ESPERADA

Adolescente
Adulto
Adulto
Jubilado

15. Caja negra: el problema del triángulo

Medio · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

Clásico de las pruebas de caja negra. Implementa tipo(a, b, c), que recibe las longitudes de tres lados enteros y devuelve "Equilátero", "Isósceles", "Escaleno" o "No es triángulo" (si algún lado es ≤ 0 o si un lado es mayor o igual que la suma de los otros dos). Entrada: tres enteros por línea.

Código de partida (java)

```
import java.util.*;

public class Main {

    static String tipo(int a, int b, int c) {
        // TODO: clasifica el triángulo
        return "";
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) {
            System.out.println(tipo(sc.nextInt(), sc.nextInt(), sc.nextInt()));
        }
    }
}
```

Ejemplo: Tipos válidos

ENTRADA

3 3 3
3 3 5
3 4 5

SALIDA ESPERADA

Equilátero
Isósceles
Escaleno

Ejemplo: Casos inválidos

ENTRADA

0 4 5
1 2 3
-3 4 5

SALIDA ESPERADA

No es triángulo
No es triángulo
No es triángulo

16. Refactoriza: extraer método

Medio · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

El cálculo original del precio estaba en un único bloque lleno de números mágicos:

$p = \text{base} * \text{unidades}$; si VIP $p -= 15 \%$; si SOCIO $p -= 5 \%$; si unidades ≥ 10 $p -= 10 \%$; $p += 21 \%$ de IVA.

Ya se ha extraído la estructura en métodos pequeños. Completa `descuentoCliente()`, `descuentoVolumen()` y el IVA de `precioFinal()` para que el comportamiento sea idéntico al original. Entrada por línea: precio base, unidades y tipo de cliente (NORMAL, SOCIO o VIP).

Código de partida (java)

```
import java.util.*;

public class Main {

    static final double IVA = 0.21;

    static double descuentoCliente(String tipo) {
        // TODO: 0.15 para VIP, 0.05 para SOCIO y 0 para el resto
        return 0;
    }

    static double descuentoVolumen(int unidades) {
        // TODO: 0.10 a partir de 10 unidades
        return 0;
    }

    static double aplicarDescuento(double importe, double descuento) {
        return importe * (1 - descuento);
    }

    static double precioFinal(double base, int unidades, String tipo) {
        double importe = base * unidades;
        importe = aplicarDescuento(importe, descuentoCliente(tipo));
        importe = aplicarDescuento(importe, descuentoVolumen(unidades));
        // TODO: aplica el IVA usando la constante
        return importe;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in).useLocale(Locale.US);
        while (sc.hasNext()) {
            double base = sc.nextDouble();
            int unidades = sc.nextInt();
            String tipo = sc.next();
            System.out.printf(Locale.US, "%.2f\n", precioFinal(base, unidades, tipo));
        }
    }
}
```

Ejemplo: Cliente normal

ENTRADA

10 2 NORMAL

SALIDA ESPERADA

24.20

Ejemplo: Socio con descuento por volumen

ENTRADA

20 12 SOCIO

SALIDA ESPERADA

248.29

17. Del diagrama de clases al código

Medio · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

Implementa el diagrama de clases: la clase abstracta Figura (nombre, area() abstracto y toString() ya hechos) y sus subclases Circulo (radio) y Rectangulo (base, altura), que heredan de ella. El programa lee figuras (C radio o R base altura), las muestra y suma sus áreas.

Figura ◁— Circulo (- radio: double)

Figura ◁— Rectangulo (- base: double, - altura: double)

Código de partida (java)

```
import java.util.*;

abstract class Figura {
    protected String nombre;

    public Figura(String nombre) { this.nombre = nombre; }

    public abstract double area();

    @Override
    public String toString() {
        return String.format(Locale.US, "%s: %.2f", nombre, area());
    }
}

class Circulo extends Figura {
    private double radio;

    public Circulo(double radio) {
        super("Círculo");
        // TODO: guarda el radio
    }

    @Override
    public double area() {
        return 0; // TODO:  $\pi \cdot r^2$ 
    }
}

class Rectangulo extends Figura {
    // TODO: atributos base y altura

    public Rectangulo(double base, double altura) {
        super("Rectángulo");
        // TODO: guarda base y altura
    }

    @Override
    public double area() {
        return 0; // TODO: base·altura
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in).useLocale(Locale.US);
        List<Figura> figuras = new ArrayList<>();
        while (sc.hasNext()) {
            String tipo = sc.next();
            if (tipo.equals("C")) figuras.add(new Circulo(sc.nextDouble()));
            else figuras.add(new Rectangulo(sc.nextDouble(), sc.nextDouble()));
        }
        double total = 0;
        for (Figura f : figuras) {
            System.out.println(f);
            total += f.area();
        }
        System.out.printf(Locale.US, "Total: %.2f\n", total);
    }
}
```

Ejemplo: Un círculo y un rectángulo

ENTRADA

```
C 1
R 3 4
```

SALIDA ESPERADA

```
Círculo: 3.14
Rectángulo: 12.00
Total: 15.14
```

Ejercicios largos

18. Validador de DNI, NIE e IBAN diseñado con clases de equivalencia

Medio · Java · 50 minutos · apuntesdam.com/ejercicios/java/validador-dni-nie-iban

Casi todas las aplicaciones de gestión piden un DNI, un NIE o un IBAN, y casi todas los validan mal: aceptan letras que no corresponden, rechazan minúsculas o se atascan con los espacios del IBAN. Los tres llevan un control matemático que permite detectar la mayoría de los errores al teclear.

La letra del DNI es el resto de dividir el número entre 23, buscado en la tabla TRWAGMYFPDXBNJZSQVHLCKE. El NIE de los extranjeros funciona igual cambiando la X, la Y y la Z iniciales por 0, 1 y 2. El IBAN usa otro método (ISO 13616): se pasan los cuatro primeros caracteres al final, cada letra se sustituye por un número (A = 10, B = 11... Z = 35) y el número resultante, de unas 26 cifras, tiene que dar resto 1 al dividirlo entre 97.

Este ejercicio es también de Entornos de Desarrollo: las pruebas no se han elegido al azar, sino con la técnica de clases de equivalencia (un caso representativo de cada grupo de entradas que el programa trata igual) y valores límite (los bordes de cada grupo). Tienes la tabla abajo: tu programa tiene que pasar todos esos casos.

Requisitos

- Cada línea es `TIPO valor`, con `TIPO` `DNI`, `NIE` o `IBAN` en mayúsculas o minúsculas. Las líneas vacías se ignoran. Para cada línea escribe `TIPO valor → válido` o `TIPO valor → no válido: motivo`, con el tipo en mayúsculas y el valor tal como venía (sin espacios al principio ni al final).
- DNI: 8 cifras y una letra (mayúscula o minúscula). Si no tiene ese formato, `formato incorrecto (8 cifras y una letra)`; si la letra no es la que toca, `la letra debería ser L`.
- NIE: X, Y o Z, 7 cifras y una letra. Si no tiene ese formato, `formato incorrecto (X, Y o Z, 7 cifras y una letra)`; si la letra no cuadra, el mismo mensaje que el DNI.
- IBAN (se ignoran los espacios y las minúsculas): si no empieza por ES, `solo se admiten IBAN españoles`; si no tiene 24 caracteres, `un IBAN español tiene 24 caracteres`; si tras ES hay algo que no sea una cifra, `después de ES solo puede haber cifras`; si el control módulo 97 no da 1, `los dígitos de control no cuadran`. Se comprueba en ese orden.
- Un tipo desconocido escribe `TIPO valor → no válido: tipo de documento desconocido`.
- Al final, `Válidos: V de T`.

Formato de la entrada

- Una línea por documento: `DNI valor`, `NIE valor` o `IBAN valor`.

Clases de equivalencia y valores límite del DNI

Clase	Caso de prueba	Resultado esperado
Válida: 8 cifras y su letra	12345678Z	válido
Válida: letra en minúscula	12345678z	válido
Límite inferior del número	00000000T	válido
Límite superior del número	99999999R	válido
No válida: letra que no toca	12345678A	la letra debería ser Z
No válida: 7 cifras	1234567Z	formato incorrecto
No válida: 9 cifras	123456789Z	formato incorrecto
No válida: sin letra	12345678	formato incorrecto

Clases de equivalencia del IBAN

Clase	Caso de prueba	Resultado esperado
Válida, con espacios	ES91 2100 0418 4502 0005 1332	válido
Válida, en minúsculas y sin espacios	es9121000418450200051332	válido
Otro país	DE89370400440532013000	solo se admiten IBAN españoles
Longitud incorrecta	ES912100041845020005133	24 caracteres
Letras tras ES	ES91 2100 0418 4502 0005 13AB	solo cifras
Control incorrecto (una cifra cambiada)	ES91 2100 0418 4502 0005 1333	no cuadran

Ejemplo: DNI y NIE

ENTRADA

DNI 12345678Z
DNI 12345678z
DNI 12345678A
DNI 1234567Z
NIE X1234567L
NIE W1234567L
PASAPORTE AB123456

SALIDA ESPERADA

DNI 12345678Z → válido
DNI 12345678z → válido
DNI 12345678A → no válido: la letra debería ser Z
DNI 1234567Z → no válido: formato incorrecto (8 cifras y una letra)
NIE X1234567L → válido
NIE W1234567L → no válido: formato incorrecto (X, Y o Z, 7 cifras y una letra)
PASAPORTE AB123456 → no válido: tipo de documento desconocido
Válidos: 3 de 7

Ejemplo: IBAN

ENTRADA

IBAN ES91 2100 0418 4502 0005 1332
iban es9121000418450200051332
IBAN DE89370400440532013000
IBAN ES912100041845020005133
IBAN ES91 2100 0418 4502 0005 13AB
IBAN ES91 2100 0418 4502 0005 1333

SALIDA ESPERADA

IBAN ES91 2100 0418 4502 0005 1332 → válido
IBAN es9121000418450200051332 → válido
IBAN DE89370400440532013000 → no válido: solo se admiten IBAN españoles
IBAN ES912100041845020005133 → no válido: un IBAN español tiene 24 caracteres
IBAN ES91 2100 0418 4502 0005 13AB → no válido: después de ES solo puede haber cifras
IBAN ES91 2100 0418 4502 0005 1333 → no válido: los dígitos de control no cuadran
Válidos: 2 de 6

Código de partida (java)

```
import java.util.Scanner;

public class Main {
    static final String LETRAS = "TRWAGMYFPDXBNJZSQVHLCKE";

    /** null si el DNI es correcto, o el motivo por el que no lo es. */
    static String validarDni(String dni) {
        // TODO
        return null;
    }

    /** El NIE se valida como un DNI cambiando X, Y y Z por 0, 1 y 2. */
    static String validarNie(String nie) {
        // TODO
        return null;
    }

    /** IBAN español: 24 caracteres y control módulo 97 (ISO 13616). */
    static String validarIban(String iban) {
        // TODO
        return null;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // TODO: valida cada línea y escribe el resultado y el resumen
    }
}
```

19. Depura el programa de notas: cinco errores escondidos

Medio · Java · 45 minutos · apuntesdam.com/ejercicios/java/depura-el-calculo-de-notas

Un compañero ha escrito el programa que calcula las notas finales de Programación y lo ha dejado para que lo revises antes de publicar las notas. Compila, no lanza excepciones y con su ejemplo de prueba (tres alumnos con notas «normales») el resultado parecía razonable. Pero varios alumnos se han quejado de que su nota no cuadra.

La regla de evaluación es: cada una de las dos prácticas vale un 20 % y el examen un 60 %; si el examen tiene menos de un 4, la nota final no puede pasar de 4 (el examen no se compensa), y la nota se redondea a un decimal. Aprueba quien tiene un 5 o más.

El código del editor tiene exactamente cinco errores, cada uno de una línea. No está permitido reescribir el programa: hay que encontrar cada error y corregirlo. Es lo que se hace en un proyecto real, donde reescribir lo que no se entiende introduce errores nuevos.

Requisitos

- La entrada tiene una línea por alumno: `nombre;práctica1;práctica2;examen`, con notas de 0 a 10 con punto decimal.
- Nota final = $0,2 \times \text{práctica1} + 0,2 \times \text{práctica2} + 0,6 \times \text{examen}$; si el examen es menor que 4, la nota final es como mucho 4; se redondea a un decimal (las mitades suben).
- Se escribe una línea por alumno, de mayor a menor nota (a igualdad, por nombre): el nombre en 20 caracteres a la izquierda y la nota con un decimal y coma en 5 caracteres a la derecha.
- Al final, `Media: X` (la media de las notas finales con un decimal) y `Aprobados: A de T`.
- Corrige los cinco errores del programa sin cambiar su estructura. Las pruebas ocultas están pensadas para detectar cada uno.

Formato de la entrada

- Una línea por alumno: `nombre;práctica1;práctica2;examen`.

Ejemplo: Una clase de cinco alumnos

ENTRADA

Ana Gil;8;9;7.5
Luis Mora;6;5;4
Eva Ruiz;10;10;3.5
Pablo Sanz;5;5;5
Marta Peña;7.5;6.5;6.2

SALIDA ESPERADA

Ana Gil	7,9
Marta Peña	6,5
Pablo Sanz	5,0
Luis Mora	4,6
Eva Ruiz	4,0
Media:	5,6
Aprobados:	3 de 5

Ejemplo: El examen de 4 sí compensa

ENTRADA

Irene Vidal;9;9;4
Jorge Lara;3;3;3.9

SALIDA ESPERADA

Irene Vidal	6,0
Jorge Lara	3,5
Media:	4,8
Aprobados:	1 de 2

Código de partida (java)

```
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Scanner;

public class Main {
    record Alumno(String nombre, double nota) {}

    /** Nota final: prácticas al 20 % cada una y examen al 60 %; con menos de un 4 en el examen, como
        mucho un 4. */
    static double notaFinal(double practica1, double practica2, double examen) {
        double nota = 0.2 * practica1 + 0.2 * practica2 + 0.6 * examen;
        if (examen <= 4) nota = Math.min(nota, 4);
        return Math.round(nota * 10) / 10;
    }

    static String uno(double x) {
        return String.format(Locale.ROOT, "%.1f", x).replace('.', ',');
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<String> lineas = new ArrayList<>();
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (!l.isEmpty()) lineas.add(l);
        }

        List<Alumno> alumnos = new ArrayList<>();
        for (int i = 1; i < lineas.size(); i++) {
            String[] p = lineas.get(i).split(";");
            alumnos.add(new Alumno(p[0], notaFinal(Double.parseDouble(p[1]), Double.parseDouble(p[2]),
                Double.parseDouble(p[3]))));
        }
        alumnos.sort((a, b) -> a.nota() != b.nota() ? Double.compare(b.nota(), a.nota()) :
            a.nombre().compareTo(b.nombre()));

        double suma = 0;
        int aprobados = 0;
        for (Alumno a : alumnos) {
            System.out.printf("%-20s%5s%n", a.nombre(), uno(a.nota()));
            suma += a.nota();
            if (a.nota() > 5) aprobados++;
        }
        System.out.println("Media: " + uno(suma / alumnos.size()));
        System.out.println("Aprobados: " + aprobados + " de " + alumnos.size());
    }
}
```

Soluciones

1. Gráfico de trabajo pendiente (burndown) de un sprint

Difícil · Desarrollo de Software · apuntesdam.com/subject/entornos/topic/desarrollo-software

Solución (java)

```
import java.util.*;

public class Main {
    record Historia(String id, int puntos, int diaTerminada) { } // diaTerminada = 0 si no se terminó

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int dias = sc.nextInt();
        List<Historia> hs = new ArrayList<>();
        while (sc.hasNext()) hs.add(new Historia(sc.next(), sc.nextInt(), sc.nextInt()));

        int total = 0;
        for (Historia h : hs) total += h.puntos();
        System.out.println("Sprint de " + dias + " días, " + total + " puntos comprometidos");

        for (int d = 0; d <= dias; d++) {
            int quedan = total;
            for (Historia h : hs) {
                if (h.diaTerminada() >= 1 && h.diaTerminada() <= d) quedan -= h.puntos(); // solo
                historias terminadas
            }
            double ideal = total - (double) total * d / dias;
            String barra = "#".repeat(quedan) + " ".repeat(total - quedan);
            String linea = String.format(Locale.US, "Día %2d | %s | %2d (ideal %.1f)", d, barra, quedan,
                ideal);
            if (quedan > Math.ceil(ideal)) linea += " (retraso)";
            System.out.println(linea);
        }

        int velocidad = 0;
        List<String> pendientes = new ArrayList<>();
        for (Historia h : hs) {
            if (h.diaTerminada() > 0) velocidad += h.puntos();
            else pendientes.add(h.id());
        }
        System.out.println("Velocidad: " + velocidad + " puntos");
        System.out.println("Vuelven al product backlog: " + (pendientes.isEmpty() ? "ninguna" :
            String.join(", ", pendientes)));
    }
}
```

Para cada día se parte del total comprometido y se restan los puntos de las historias terminadas ese día o antes; las no terminadas (día 0) nunca restan. La barra son tantos # como puntos quedan, rellena con espacios hasta el total para que las columnas queden alineadas.

El ideal es una recta que baja de T a 0: $T - T \cdot d / D$, con la división en `double`. Comparar con el ideal redondeado hacia arriba (`Math.ceil`) evita marcar retraso por una fracción de punto, ya que las historias no se terminan a medias.

2. Planificación con dependencias y camino crítico

Muy difícil · Desarrollo de Software · apuntesdam.com/subject/entornos/topic/desarrollo-software

Solución (java)

```
import java.util.*;

public class Main {
    record Tarea(String id, int duracion, List<String> previas) { }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Map<String, Tarea> tareas = new LinkedHashMap<>();
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.isEmpty()) continue;
            // Formato: ID duración [previa1,previa2,...]
            String[] p = l.split("\\s+");
            List<String> previas = p.length > 2 ? List.of(p[2].split(",")) : List.of();
            tareas.put(p[0], new Tarea(p[0], Integer.parseInt(p[1]), previas));
        }

        // Quién depende de cada tarea y cuántas previas le faltan
        Map<String, List<String>> dependientes = new LinkedHashMap<>();
        Map<String, Integer> pendientes = new HashMap<>();
        for (Tarea t : tareas.values()) dependientes.put(t.id(), new ArrayList<>());
        for (Tarea t : tareas.values()) {
            for (String p : t.previas()) {
                if (!tareas.containsKey(p)) {
                    System.out.println("Error: " + t.id() + " depende de " + p + ", que no existe");
                    return;
                }
                dependientes.get(p).add(t.id());
            }
            pendientes.put(t.id(), t.previas().size());
        }

        // Algoritmo de Kahn
        Deque<String> cola = new ArrayDeque<>();
        for (Tarea t : tareas.values()) if (t.previas().isEmpty()) cola.add(t.id());
        List<String> orden = new ArrayList<>();
        while (!cola.isEmpty()) {
            String t = cola.poll();
            orden.add(t);
            for (String d : dependientes.get(t)) {
                pendientes.merge(d, -1, Integer::sum);
                if (pendientes.get(d) == 0) cola.add(d);
            }
        }
        if (orden.size() < tareas.size()) {
            System.out.println("Error: hay dependencias circulares");
            return;
        }

        // Fechas tempranas (hacia delante)
        Map<String, Integer> inicio = new HashMap<>(), fin = new HashMap<>();
        int duracion = 0;
        for (String id : orden) {
            int ini = 0;
            for (String p : tareas.get(id).previas()) ini = Math.max(ini, fin.get(p));
            inicio.put(id, ini);
            fin.put(id, ini + tareas.get(id).duracion());
            duracion = Math.max(duracion, fin.get(id));
        }

        // Fin más tardío (hacia atrás)
        Map<String, Integer> finTardio = new HashMap<>();
        for (int i = orden.size() - 1; i >= 0; i--) {
            String id = orden.get(i);
            int ft = duracion;
            for (String d : dependientes.get(id)) ft = Math.min(ft, finTardio.get(d) -
                tareas.get(d).duracion());
        }
    }
}
```

```

        finTardio.put(id, ft);
    }

    System.out.println("Orden: " + String.join(" ", orden));
    List<String> critico = new ArrayList<>();
    for (String id : orden) {
        int holgura = finTardio.get(id) - fin.get(id);
        System.out.printf("%-3s inicio %2d fin %2d holgura %d\n", id, inicio.get(id), fin.get(id),
            holgura);
        if (holgura == 0) critico.add(id);
    }
    System.out.println("Duración del proyecto: " + duracion + " días");
    System.out.println("Camino crítico: " + String.join(" → ", critico));
}
}

```

El algoritmo de Kahn lleva la cuenta de cuántas previas le faltan a cada tarea. Empiezan las que no tienen ninguna; cada vez que una sale de la cola, sus dependientes tienen una previa menos y, al llegar a 0, entran en la cola. Si al final no han salido todas, las que quedan se esperan unas a otras: hay un ciclo.

Con el orden topológico, las fechas se calculan en dos pasadas. Hacia delante, cada tarea empieza cuando termina la última de sus previas. Hacia atrás, cada tarea debe terminar antes de que tenga que empezar la más urgente de sus dependientes (su fin tardío menos su duración); las que no tienen dependientes pueden acabar al final del proyecto.

La holgura es cuánto puede retrasarse una tarea sin retrasar el proyecto. Las de holgura 0 forman el camino crítico: cualquier retraso en ellas se nota en la fecha final.

3. Depura el resumen de notas

Difícil · Instalación y uso de IDEs · apuntesdam.com/subject/entornos/topic/ides

Solución (java)

```

import java.util.*;

public class Main {
    static String resumen(int[] notas) {
        int suma = 0, aprobados = 0, max = notas[0], min = notas[0]; // 3: empezar con una nota real,
        no con 0
        for (int i = 0; i < notas.length; i++) { // 1: el primer índice es 0
            suma += notas[i];
            if (notas[i] >= 5) aprobados++; // 2: un 5 también aprueba
            if (notas[i] > max) max = notas[i];
            if (notas[i] < min) min = notas[i];
        }
        double media = (double) suma / notas.length; // 4: división con decimales
        return String.format(Locale.US, "media %.2f, aprobados %d de %d, máxima %d, mínima %d",
            media, aprobados, notas.length, max, min);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (l.isEmpty()) continue;
            int[] notas = Arrays.stream(l.split("\\s+")).mapToInt(Integer::parseInt).toArray();
            System.out.println(Arrays.toString(notas) + " → " + resumen(notas));
        }
    }
}

```

Los cuatro errores se ven en cuanto se observa el bucle con el depurador: (1) empieza en `i = 1`, así que la primera nota nunca se cuenta; (2) `> 5` deja fuera el 5, que es un aprobado; (3) `min` empieza en 0 y ninguna nota es menor que 0, así que la mínima siempre sale 0; y (4) `suma / notas.length` es una división entera y pierde los decimales.

La corrección de `min` (y de `max`, por coherencia) es empezar con una nota real, `notas[0]`. Con un punto de ruptura dentro del bucle se ve enseguida que `min` no cambia nunca, y en la primera parada, que `i` ya vale 1.

4. Depura una búsqueda binaria

Muy difícil · Instalación y uso de IDEs · apuntesdam.com/subject/entornos/topic/ides

Solución (java)

```
import java.util.*;

public class Main {
    // Primera posición i tal que a[i] >= x (a.length si no hay ninguna). El array está ordenado.
    static int primeraMayorOIgual(int[] a, int x) {
        int lo = 0, hi = a.length;           // hi = a.length: la respuesta puede ser «ninguna»
        while (lo < hi) {
            int medio = lo + (hi - lo) / 2;    // sin desbordamiento, a diferencia de (lo + hi) / 2
            if (a[medio] < x) lo = medio + 1;  // medio no es la respuesta: se descarta
            else hi = medio;                  // medio puede ser la respuesta: se conserva
        }
        return lo;
    }

    static int contar(int[] a, int x) {
        return primeraMayorOIgual(a, x + 1) - primeraMayorOIgual(a, x);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[] a =
            Arrays.stream(sc.nextLine().trim().split("\\s+")).mapToInt(Integer::parseInt).toArray();
        while (sc.hasNextInt()) {
            int x = sc.nextInt();
            System.out.println("x=" + x + ": primera posición >= x → " + primeraMayorOIgual(a, x) + ",
                apariciones → " + contar(a, x));
        }
    }
}
```

La clave de una búsqueda binaria correcta es el invariante: la respuesta está siempre en el intervalo $[lo, hi]$. Como puede ser `a.length` (todos menores), `hi` tiene que empezar en `a.length` y no en `a.length - 1`; ese era el motivo de que nunca devolviera `a.length`.

En cada vuelta, si `a[medio] < x` la respuesta está estrictamente a la derecha (`lo = medio + 1`; con `lo = medio` el intervalo podía no encogerse y se quedaba en bucle); si no, `medio` es candidata y se conserva con `hi = medio` (con `medio - 1` se perdía). El bucle termina cuando `lo == hi`.

$(lo + hi) / 2$ se desborda si la suma pasa de `Integer.MAX_VALUE`; `lo + (hi - lo) / 2` calcula lo mismo sin sumar dos índices grandes.

5. Valores límite: casos que detectan errores

Difícil · Diseño y realización de pruebas · apuntesdam.com/subject/entornos/topic/disenio-pruebas

Solución (java)

```
import java.util.*;
import java.util.function.DoubleUnaryOperator;

public class Main {
    // Tarifa de envío según el peso en kg. -1 significa «no admitido».
    // peso <= 0      → -1
    // 0 < peso <= 1  → 4.5
    // 1 < peso <= 5  → 7
    // 5 < peso <= 20 → 12
    // peso > 20     → -1
    static double tarifa(double peso) {
        if (peso <= 0) return -1;
        if (peso <= 1) return 4.5;
        if (peso <= 5) return 7;
        if (peso <= 20) return 12;
        return -1;
    }

    // Cada límite se prueba justo en él y justo por encima
    static final double[][] CASOS = {
        {0, -1},          // límite inferior: 0 no se admite (detecta M4)
        {0.01, 4.5},
        {1, 4.5},         // 1 kg aún es el primer tramo (detecta M1)
        {1.01, 7},        // primer valor del segundo tramo (detecta M6)
        {5, 7},           // 5 kg aún es el segundo tramo (detecta M2)
        {5.01, 12},
        {20, 12},         // 20 kg aún se admite (detecta M3)
        {20.01, -1},     // por encima de 20 no se admite (detecta M5)
    };

    // ---- No modifique lo que sigue: versiones con un error cada una («mutantes»)
    static final Map<String, DoubleUnaryOperator> MUTANTES = new LinkedHashMap<>();
    static {
        MUTANTES.put("M1 (< 1 en vez de <= 1)", p -> p <= 0 ? -1 : p < 1 ? 4.5 : p <= 5 ? 7 : p <= 20 ? 12 : -1);
        MUTANTES.put("M2 (< 5 en vez de <= 5)", p -> p <= 0 ? -1 : p <= 1 ? 4.5 : p < 5 ? 7 : p <= 20 ? 12 : -1);
        MUTANTES.put("M3 (< 20 en vez de <= 20)", p -> p <= 0 ? -1 : p <= 1 ? 4.5 : p <= 5 ? 7 : p < 20 ? 12 : -1);
        MUTANTES.put("M4 (< 0 en vez de <= 0)", p -> p < 0 ? -1 : p <= 1 ? 4.5 : p <= 5 ? 7 : p <= 20 ? 12 : -1);
        MUTANTES.put("M5 (sin límite superior)", p -> p <= 0 ? -1 : p <= 1 ? 4.5 : p <= 5 ? 7 : 12);
        MUTANTES.put("M6 (7.5 en vez de 7)", p -> p <= 0 ? -1 : p <= 1 ? 4.5 : p <= 5 ? 7.5 : p <= 20 ? 12 : -1);
    }

    public static void main(String[] args) {
        String fallo = null;
        for (double[] c : CASOS)
            if (tarifa(c[0]) != c[1] && fallo == null) fallo = "tarifa(" + c[0] + ") da " + tarifa(c[0]) + ", no " + c[1];
        System.out.println("Casos correctos con el original: " + (fallo == null ? "sí" : "no (" + fallo + ")"));
        List<String> vivos = new ArrayList<>();
        for (var m : MUTANTES.entrySet()) {
            boolean detectado = false;
            for (double[] c : CASOS) if (m.getValue().applyAsDouble(c[0]) != c[1]) detectado = true;
            if (!detectado) vivos.add(m.getKey());
        }
        System.out.println("Mutantes detectados: " + (MUTANTES.size() - vivos.size()) + " de " + MUTANTES.size());
        System.out.println("Sobreviven: " + (vivos.isEmpty() ? "ninguno" : String.join(", ", vivos)));
    }
}
```

Un $<$ en lugar de $\leq$ solo cambia el resultado exactamente en el límite, así que un caso en mitad del tramo (como el 3) no detecta nada. Por eso se prueba cada frontera dos veces: justo en el límite (0, 1, 5, 20) y justo por encima (0,01, 1,01, 5,01, 20,01).

Cada caso se explica por el error que detecta: el 0 caza < 0 , el 1 caza < 1 , el 20,01 caza el límite superior olvidado y cualquier peso del segundo tramo caza el precio equivocado. Es la idea del análisis de valores límite: pocos casos, pero elegidos donde los errores son probables.

6. Matar mutantes con varias condiciones

Muy difícil · Diseño y realización de pruebas · apuntesdam.com/subject/entornos/topic/disenio-pruebas

Solución (java)

```
import java.util.*;

public class Main {
    interface Precio { double calcular(int edad, boolean socio, boolean finDeSemana); }

    // Precio de la entrada a un museo:
    //  edad < 0 → -1 (error)
    //  base 10 €; los fines de semana, 12 €
    //  menores de 12: 50 % de descuento; desde 65: 40 % de descuento
    //  los socios tienen además 2 € menos sobre el precio ya descontado (nunca por debajo de 0)
    static final Precio ORIGINAL = (edad, socio, finde) -> {
        if (edad < 0) return -1;
        double p = finde ? 12 : 10;
        if (edad < 12) p *= 0.5;
        else if (edad >= 65) p *= 0.6;
        if (socio) p = Math.max(0, p - 2);
        return p;
    };

    // {edad, socio (0/1), fin de semana (0/1), precio esperado}
    static final double[][] CASOS = {
        {30, 0, 0, 10}, // adulto entre semana
        {30, 0, 1, 12}, // adulto en fin de semana
        {-1, 0, 0, -1}, // edad no válida
        {0, 0, 0, 5}, // 0 años es válido (detecta M3)
        {11, 0, 0, 5}, // último año de niño
        {12, 0, 0, 10}, // 12 ya no es niño (detecta M1)
        {64, 0, 0, 10}, // último año sin descuento de mayor
        {65, 0, 0, 6}, // 65 ya tiene el 40 % (detecta M2 y M8)
        {10, 0, 1, 6}, // niño en fin de semana: 50 % de 12 (detecta M6)
        {10, 1, 0, 3}, // niño socio: 5 - 2 (detecta M4 y M7)
        {70, 1, 1, 5.2}, // mayor socio en fin de semana: 7,2 - 2
        {30, 1, 0, 8}, // adulto socio
    };

    // ---- No modifique lo que sigue
    static final Map<String, Precio> MUTANTES = new LinkedHashMap<>();
    static {
        MUTANTES.put("M1 (<= 12)", (e, s, f) -> e < 0 ? -1 : aplicar(f ? 12 : 10, e <= 12, e >= 65, s));
        MUTANTES.put("M2 (> 65)", (e, s, f) -> e < 0 ? -1 : aplicar(f ? 12 : 10, e < 12, e > 65, s));
        MUTANTES.put("M3 (<= 0 es error)", (e, s, f) -> e <= 0 ? -1 : aplicar(f ? 12 : 10, e < 12, e >= 65, s));
        MUTANTES.put("M4 (socio antes del %)", (e, s, f) -> {
            if (e < 0) return -1;
            double p = (f ? 12 : 10) - (s ? 2 : 0);
            return e < 12 ? p * 0.5 : e >= 65 ? p * 0.6 : p;
        });
        MUTANTES.put("M5 (sin tope en 0)", (e, s, f) -> {
            if (e < 0) return -1;
            double p = f ? 12 : 10;
            if (e < 12) p *= 0.5; else if (e >= 65) p *= 0.6;
            return s ? p - 2 : p;
        });
        MUTANTES.put("M6 (ignora el fin de semana con descuento)", (e, s, f) -> e < 0 ? -1 : aplicar((f
            && e >= 12 && e < 65) ? 12 : 10, e < 12, e >= 65, s));
        MUTANTES.put("M7 (el socio solo descuenta a adultos)", (e, s, f) -> e < 0 ? -1 : aplicar(f ? 12 :
            10, e < 12, e >= 65, s && e >= 12 && e < 65));
        MUTANTES.put("M8 (65 o más: 50 %)", (e, s, f) -> {
            if (e < 0) return -1;
            double p = f ? 12 : 10;
            if (e < 12 || e >= 65) p *= 0.5;
            if (s) p = Math.max(0, p - 2);
            return p;
        });
    }
}
```

```

static double aplicar(double p, boolean nino, boolean mayor, boolean socio) {
    if (nino) p *= 0.5; else if (mayor) p *= 0.6;
    if (socio) p = Math.max(0, p - 2);
    return p;
}

static boolean igual(double a, double b) { return Math.abs(a - b) < 1e-9; }

public static void main(String[] args) {
    String fallo = null;
    for (double[] c : CASOS) {
        double r = ORIGINAL.calcular((int) c[0], c[1] == 1, c[2] == 1);
        if (!igual(r, c[3]) && fallo == null) fallo = Arrays.toString(c) + " da " + r;
    }
    System.out.println("Casos correctos con el original: " + (fallo == null ? "sí" : "no (" + fallo +
        ")"));
    List<String> vivos = new ArrayList<>();
    for (var m : MUTANTES.entrySet()) {
        boolean detectado = false;
        for (double[] c : CASOS)
            if (!igual(m.getValue().calcular((int) c[0], c[1] == 1, c[2] == 1), c[3])) detectado =
                true;
        if (!detectado) vivos.add(m.getKey());
    }
    System.out.println("Mutantes detectados: " + (MUTANTES.size() - vivos.size()) + " de " +
        MUTANTES.size());
    System.out.println("Sobreviven: " + (vivos.isEmpty() ? "ninguno" : String.join(", ", vivos)));
}
}

```

Cada límite de edad se prueba a los dos lados (11 y 12, 64 y 65, -1 y 0), y cada regla se combina con otra que la haga visible: el fin de semana con un niño (si se ignora, sale 5 en vez de 6) y el descuento de socio con un niño (si se aplica antes del 50 %, sale 4 en vez de 3; si solo se aplica a adultos, sale 5).

El mutante equivalente es M5, el que quita el tope de 0 €. El precio más bajo posible antes del descuento de socio es 5 € (un niño entre semana), y $5 - 2 = 3$ nunca es negativo, así que `Math.max(0, ...)` no cambia nada con ninguna entrada. Ningún caso puede detectarlo: no es un fallo de las pruebas, es código que sobra.

7. Un analizador de código en miniatura

Difícil · Optimización y Documentación · apuntesdam.com/subject/entornos/topic/optimizacion-documentacion

Solución (java)

```
import java.util.*;
import java.util.regex.*;

public class Main {
    static final int MAX_LINEAS_METODO = 8, MAX_PARAMETROS = 3;
    static final Pattern METODO = Pattern.compile(
        "^\\s*(public|private|protected|static|\\s)+[\\w<>\\\\[\\\\]]+\\s+(\\w+)\\s*\\\\(((^)*))\\\\s*\\\\s*\\\\s*"
    );
    static final Pattern NUMERO = Pattern.compile("\\b\\d+(\\.\\d+)?\\b");

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<String> lineas = new ArrayList<>();
        while (sc.hasNextLine()) lineas.add(sc.nextLine());
        List<String> avisos = analizar(lineas);
        avisos.forEach(System.out::println);
        System.out.println(avisos.size() + (avisos.size() == 1 ? " aviso" : " avisos"));
    }

    // La línea sin cadenas ni comentarios //
    static String codigo(String linea) {
        String s = linea.replaceAll("\\\"(\\\\\\\\.|[^\"\\\\\\\\])*\\""", "\\\"\\\"");
        int c = s.indexOf("/");
        return c >= 0 ? s.substring(0, c) : s;
    }

    static List<String> analizar(List<String> lineas) {
        List<String> avisos = new ArrayList<>();
        String metodo = null; // método en el que estamos
        int inicio = 0, llaves = 0;
        for (int i = 0; i < lineas.size(); i++) {
            int n = i + 1;
            String linea = codigo(lineas.get(i));
            Matcher m = METODO.matcher(linea);
            if (m.find()) {
                String nombre = m.group(2), params = m.group(3).trim();
                if (!nombre.matches("[a-z][a-zA-Z0-9]*")) avisos.add("Línea " + n + ": el método «" + nombre + "» no sigue lowerCamelCase");
                int p = params.isEmpty() ? 0 : params.split(",").length;
                if (p > MAX_PARAMETROS) avisos.add("Línea " + n + ": «" + nombre + "» tiene " + p + " parámetros (máximo " + MAX_PARAMETROS + ")");
                if (metodo == null) {
                    metodo = nombre;
                    inicio = n;
                    llaves = 0;
                }
            }
            if (!linea.contains("final")) {
                Matcher num = NUMERO.matcher(linea);
                while (num.find()) {
                    if (!num.group().equals("0") && !num.group().equals("1")) avisos.add("Línea " + n + ": número mágico " + num.group());
                }
            }
            if (metodo != null) {
                for (char c : linea.toCharArray()) {
                    if (c == '{') llaves++;
                    if (c == '}') llaves--;
                }
                if (llaves == 0) { // se cierra el método
                    int largo = n - inicio + 1;
                    if (largo > MAX_LINEAS_METODO) avisos.add("Línea " + inicio + ": «" + metodo + "» mide " + largo + " líneas (máximo " + MAX_LINEAS_METODO + ")");
                    metodo = null;
                }
            }
        }
    }
}
```

```
    }  
    return avisos;  
  }  
}
```

Antes de aplicar las reglas, cada línea se limpia: las cadenas se vacían con una expresión regular y se corta lo que va tras `//`. Así ni el 99 de «Pedido de 99 unidades» cuenta como número mágico ni una llave dentro de un texto descuadra el recuento.

Las reglas 1 y 2 se aplican al reconocer una cabecera de método; la 3 a cada línea que no declare una constante, buscando números con `\b\d+(\.\d+)?\b` (los `\b` evitan coger la cifra de un nombre como `x1`).

Para la regla 4 se recuerda el método actual y se cuentan las llaves desde su cabecera: cuando el contador vuelve a 0, el método ha terminado y se sabe cuántas líneas mide. Es la misma técnica que usan los analizadores reales, aunque ellos trabajan sobre el árbol sintáctico en lugar del texto.

8. Un Git en miniatura: base de fusión

Muy difícil · Optimización y Documentación · apuntesdam.com/subject/entornos/topic/optimizacion-documentacion

Solución (java)

```
import java.util.*;

public class Main {
    static final Map<String, List<String>> padres = new LinkedHashMap<>(); // commit → sus padres
    static final Map<String, String> ramas = new TreeMap<>(); // rama → commit al que apunta

    // Todos los antepasados de un commit, incluido él mismo
    static Set<String> antepasados(String c) {
        Set<String> vistos = new HashSet<>();
        Deque<String> pendientes = new ArrayDeque<>();
        if (c != null) pendientes.push(c);
        while (!pendientes.isEmpty()) {
            String x = pendientes.pop();
            if (!vistos.add(x)) continue; // ya visitado (en un grafo con fusiones se llega por varios caminos)
            for (String p : padres.getOrDefault(x, List.of())) pendientes.push(p);
        }
        return vistos;
    }

    // Antepasado común «más reciente» (el que no es antepasado de otro antepasado común); null si no hay
    static String baseDeFusion(String a, String b) {
        Set<String> comunes = antepasados(a);
        comunes.retainAll(antepasados(b));
        for (String c : padres.keySet()) { // en orden de creación
            if (!comunes.contains(c)) continue;
            boolean esAntepasadoDeOtro = false;
            for (String otro : comunes) {
                if (!otro.equals(c) && antepasados(otro).contains(c)) esAntepasadoDeOtro = true;
            }
            if (!esAntepasadoDeOtro) return c;
        }
        return null;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] t = sc.nextLine().trim().split("\\s+");
            switch (t[0]) {
                case "COMMIT" -> padres.put(t[1], t.length > 2 ? List.of(t[2].split(",")) : List.of());
                case "RAMA" -> ramas.put(t[1], t[2]);
                case "BASE" -> System.out.println("merge-base " + t[1] + " " + t[2] + " → " +
                    baseDeFusion(ramas.get(t[1]), ramas.get(t[2])));
                case "FUSIONAR" -> { // FUSIONAR destino origen: ¿qué haría git merge origen estando en destino?
                    String d = ramas.get(t[1]), o = ramas.get(t[2]);
                    String base = baseDeFusion(d, o);
                    String r;
                    if (o.equals(base)) r = "ya actualizado (nada que fusionar)";
                    else if (d.equals(base)) r = "fast-forward: " + t[1] + " pasa de " + d + " a " + o;
                    else r = "fusión de tres vías (base " + base + "): se crea un commit de fusión";
                    System.out.println("git merge " + t[2] + " en " + t[1] + " → " + r);
                }
            }
        }
    }
}
```

antepasados es un recorrido del grafo de commits siguiendo los padres, con una pila y un conjunto de visitados. El conjunto es necesario porque, tras una fusión, un commit tiene dos padres y a un mismo antepasado se puede llegar por dos caminos.

Los antepasados comunes son la intersección de los dos conjuntos (`retainAll`). De ellos, la base de fusión es el más reciente, es decir, el que no es antepasado de ningún otro común. Con eso, el `main` ya distingue los tres casos de `git merge` : si la base es el origen no hay nada nuevo, si es el destino basta con mover la rama, y si no, hay trabajo en los dos lados.

9. Dobles de prueba escritos a mano

Difícil · Pruebas con JUnit y Mockito · apuntesdam.com/subject/entornos/topic/pruebas-junit

Solución (java)

```
import java.util.*;

interface Inventario {
    int stock(String producto);
    void descontar(String producto, int unidades);
}

interface Correo {
    void enviar(String destinatario, String asunto);
}

// Clase que se quiere probar: no sabe si habla con una base de datos real o con un doble
class ServicioPedidos {
    private final Inventario inventario;
    private final Correo correo;

    ServicioPedidos(Inventario inventario, Correo correo) {
        this.inventario = inventario;
        this.correo = correo;
    }

    boolean pedir(String cliente, String producto, int unidades) {
        if (unidades <= 0) return false;
        if (inventario.stock(producto) < unidades) {
            correo.enviar("almacen@tienda.es", "Sin stock: " + producto);
            return false;
        }
        inventario.descontar(producto, unidades);
        correo.enviar(cliente, "Pedido confirmado: " + unidades + " x " + producto);
        return true;
    }
}

// Stub (devuelve lo que fija la prueba) y a la vez espía (anota las llamadas)
class InventarioFalso implements Inventario {
    private final int stockFijo;
    private final List<String> descuentos = new ArrayList<>();

    InventarioFalso(int stockFijo) { this.stockFijo = stockFijo; }
    public int stock(String producto) { return stockFijo; }
    public void descontar(String producto, int unidades) { descuentos.add(producto + " -" + unidades); }
    List<String> descuentos() { return descuentos; }
}

// Espía: no envía nada, solo recuerda lo que se le pidió
class CorreoEspia implements Correo {
    private final List<String> enviados = new ArrayList<>();

    public void enviar(String destinatario, String asunto) { enviados.add(destinatario + " | " + asunto); }
    List<String> enviados() { return enviados; }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] t = sc.nextLine().trim().split(" "); // stock cliente producto unidades
            InventarioFalso inv = new InventarioFalso(Integer.parseInt(t[0]));
            CorreoEspia correo = new CorreoEspia();
            boolean ok = new ServicioPedidos(inv, correo).pedir(t[1], t[2], Integer.parseInt(t[3]));
            System.out.println("pedir(" + t[1] + ", " + t[2] + ", " + t[3] + ") con stock " + t[0] + " → " + ok);
            System.out.println(" descuentos: " + inv.descuentos());
            System.out.println(" correos: " + correo.enviados());
        }
    }
}
```

```
}  
}
```

Los dobles implementan las mismas interfaces que los objetos reales, así que `ServicioPedidos` los usa sin enterarse. `InventarioFalso` devuelve siempre el stock que fija la prueba (eso es un stub) y anota cada llamada a `descontar` (eso es un espía); `CorreoEspia` solo anota los envíos.

Con las listas que guardan, la prueba puede comprobar no solo lo que devuelve el método, sino también con quién habló y cómo: que no se descontó nada si no había stock, o que se avisó al almacén. Es lo que Mockito hace por ti con `when(...).thenReturn(...)` y `verify(...)`.

10. Tu propio JUnit con reflexión

Muy difícil · Pruebas con JUnit y Mockito · apuntesdam.com/subject/entornos/topic/pruebas-junit

Solución (java)

```
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

// Anotaciones propias, como @Test y @BeforeEach de JUnit
@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.METHOD) @interface Prueba { }
@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.METHOD) @interface Antes { }

class FalloAsercion extends RuntimeException {
    FalloAsercion(String m) { super(m); }
}

class Afirmar {
    static void iguales(Object esperado, Object real) {
        if (!Objects.equals(esperado, real)) throw new FalloAsercion("se esperaba <" + esperado + "> pero fue <" + real + ">");
    }
    static void cierto(boolean c, String mensaje) {
        if (!c) throw new FalloAsercion(mensaje);
    }
}

// ---- Las pruebas de ejemplo (no las modifiques)
class PilaTest {
    private Deque<Integer> pila;

    @Antes void preparar() { pila = new ArrayDeque<>(); pila.push(1); pila.push(2); }

    @Prueba void cimaEsElUltimo() { Afirmar.iguales(2, pila.peek()); }
    @Prueba void sacarReduce() { pila.pop(); Afirmar.iguales(1, pila.size()); }
    @Prueba void cadaPruebaEmpiezaLimpia() { Afirmar.iguales(2, pila.size()); }
    @Prueba void fallaAProposito() { Afirmar.iguales(3, pila.size()); }
    @Prueba void lanzaExcepcion() { new ArrayDeque<Integer>().pop(); }
    void noEsUnaPrueba() { throw new IllegalStateException("no debería ejecutarse"); }
}

public class Main {
    // Ejecuta todos los métodos @Prueba de la clase, en orden alfabético, con una instancia nueva cada vez
    static void ejecutar(Class<?> clase) throws Exception {
        List<Method> pruebas = new ArrayList<>(), antes = new ArrayList<>();
        for (Method m : clase.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Prueba.class)) pruebas.add(m);
            if (m.isAnnotationPresent(Antes.class)) antes.add(m);
        }
        pruebas.sort(Comparator.comparing(Method::getName));

        int correctas = 0, fallidas = 0, errores = 0;
        for (Method prueba : pruebas) {
            Constructor<?> constructor = clase.getDeclaredConstructor();
            constructor.setAccessible(true);
            Object instancia = constructor.newInstance(); // una instancia nueva por prueba
            try {
                for (Method a : antes) {
                    a.setAccessible(true);
                    a.invoke(instancia);
                }
                prueba.setAccessible(true);
                prueba.invoke(instancia);
                System.out.println("✓ " + prueba.getName());
                correctas++;
            } catch (InvocationTargetException e) {
                Throwable causa = e.getCause(); // la excepción que lanzó la prueba
                if (causa instanceof FalloAsercion) {
                    System.out.println("X " + prueba.getName() + ": " + causa.getMessage());
                    fallidas++;
                }
            }
        }
    }
}
```

```

        } else {
            System.out.println("!! " + prueba.getName() + ": " +
                causa.getClass().getSimpleName());
            errores++;
        }
    }
}
System.out.println("Pruebas: " + pruebas.size() + ", correctas: " + correctas + ", fallidas: " +
    fallidas + ", con error: " + errores);
}

public static void main(String[] args) throws Exception {
    ejecutar(PilaTest.class);
}
}

```

`getDeclaredMethods()` devuelve los métodos de la clase y `isAnnotationPresent` dice si llevan una anotación. Esto solo funciona porque las anotaciones tienen `@Retention(RUNTIME)`: si no, el compilador no las guardaría en la clase y la reflexión no las vería.

Cada prueba recibe una instancia recién creada, y sobre ella se ejecutan los métodos `@Antes` y la prueba con `invoke`. Por eso `cadaPruebaEmpiezaLimpia` pasa aunque `sacarReduce` haya quitado un elemento: no comparten la pila.

Cuando un método llamado por reflexión lanza una excepción, `invoke` la envuelve en una `InvocationTargetException`; la original está en `getCause()`. Su tipo distingue un fallo (una aserción que no se cumple) de un error (cualquier otra excepción), igual que JUnit.

11. Del diagrama de estados al código

Difícil · Diagramas UML · apuntesdam.com/subject/entornos/topic/diagramas-uml

Solución (java)

```
import java.util.*;

public class Main {
    // Diagrama de estados de un pedido (estado inicial: CARRITO)
    // CARRITO --pagar--> PAGADO
    // CARRITO --cancelar--> CANCELADO
    // PAGADO --enviar--> ENVIADO
    // PAGADO --cancelar--> CANCELADO [acción: devolver el dinero]
    // ENVIADO --entregar--> ENTREGADO
    // ENVIADO --perder--> PAGADO [acción: reenviar; como mucho 2 veces, después CANCELADO con devolución]
    // ENTREGADO y CANCELADO son estados finales
    enum Estado { CARRITO, PAGADO, ENVIADO, ENTREGADO, CANCELADO }

    static Estado estado = Estado.CARRITO;
    static int perdidas = 0;

    // Aplica un evento y devuelve el texto que describe lo ocurrido
    static String aplicar(String evento) {
        Estado siguiente = null;
        String accion = null;
        switch (estado) {
            case CARRITO -> {
                if (evento.equals("pagar")) siguiente = Estado.PAGADO;
                else if (evento.equals("cancelar")) siguiente = Estado.CANCELADO;
            }
            case PAGADO -> {
                if (evento.equals("enviar")) siguiente = Estado.ENVIADO;
                else if (evento.equals("cancelar")) { siguiente = Estado.CANCELADO; accion = "devolver el dinero"; }
            }
            case ENVIADO -> {
                if (evento.equals("entregar")) siguiente = Estado.ENTREGADO;
                else if (evento.equals("perder")) {
                    perdidas++;
                    if (perdidas < 3) { siguiente = Estado.PAGADO; accion = "reenviar"; }
                    else { siguiente = Estado.CANCELADO; accion = "devolver el dinero"; }
                }
            }
            default -> { } // ENTREGADO y CANCELADO: estados finales, sin transiciones
        }
        if (siguiente == null) return "transición no permitida (sigue en " + estado + ")";
        estado = siguiente;
        return estado + (accion != null ? " [" + accion + "]" : "");
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNext()) {
            String ev = sc.next();
            Estado antes = estado;
            String r = aplicar(ev);
            System.out.println(antes + " --" + ev + "--> " + r);
        }
        System.out.println("Estado final: " + estado);
    }
}
```

Cada flecha del diagrama se convierte en una rama: el `switch` elige el estado actual y, dentro, cada `if` es un evento que sale de él. Si ninguna rama asigna `siguiente`, el evento no tiene transición desde ese estado y no cambia nada; los estados finales simplemente no tienen ramas.

La condición de la pérdida (como mucho dos reenvíos) es lo que en UML se llama una guarda: la misma transición lleva a un estado u otro según el contador `perdidas`, que forma parte del estado del objeto aunque no aparezca como un estado del diagrama.

12. Ingeniería inversa: de las clases al diagrama

Muy difícil · Diagramas UML · apuntesdam.com/subject/entornos/topic/diagramas-uml

Solución (java)

```
import java.lang.reflect.*;
import java.util.*;

// ---- Modelo de ejemplo (no lo modifiques)
abstract class Persona {
    protected String nombre;
    private String dni;
    public String getNombre() { return nombre; }
}
interface Evaluable { double nota(); }
class Alumno extends Persona implements Evaluable {
    private int curso;
    private List<Matricula> matriculas;
    public double nota() { return 0; }
    public void matricular(Asignatura a) { }
}
class Profesor extends Persona {
    private double salario;
    private List<Asignatura> imparte;
}
class Asignatura {
    private String codigo;
    int horas;
    private Profesor titular;
}
class Matricula {
    private Asignatura asignatura;
    private double nota;
}

public class Main {
    static final List<Class<?>> MODELO = List.of(Persona.class, Evaluable.class, Alumno.class,
        Profesor.class, Asignatura.class, Matricula.class);

    static String visibilidad(int mod) {
        return Modifier.isPublic(mod) ? "+" : Modifier.isPrivate(mod) ? "-" : Modifier.isProtected(mod) ?
            "#" : "~";
    }

    // Genera el diagrama de clases en PlantUML a partir de las clases compiladas
    static String plantUml(List<Class<?>> clases) {
        StringBuilder sb = new StringBuilder("@startuml\n");
        List<String> relaciones = new ArrayList<>();
        for (Class<?> c : clases) {
            String nombre = c.getSimpleName();
            String tipo = c.isInterface() ? "interface" : Modifier.isAbstract(c.getModifiers()) ?
                "abstract class" : "class";
            sb.append(tipo).append(' ').append(nombre).append(" {\n");

            Field[] campos = c.getDeclaredFields();
            Arrays.sort(campos, Comparator.comparing(Field::getName));
            for (Field f : campos) {
                if (f.isSynthetic()) continue;
                Class<?> t = f.getType();
                if (clases.contains(t)) { // referencia a otra clase
                    del modelo
                    relaciones.add(nombre + " --> \"1\" " + t.getSimpleName() + " : " + f.getName());
                } else if (List.class.isAssignableFrom(t) && f.getGenericType() instanceof
                    ParameterizedType pt
                    && pt.getActualTypeArguments()[0] instanceof Class<?> e && clases.contains(e)) {
                    relaciones.add(nombre + " --> \"*\n\" " + e.getSimpleName() + " : " + f.getName());
                } else {
                    sb.append(" ").append(visibilidad(f.getModifiers())).append(f.getName())
                        .append(" : ").append(t.getSimpleName()).append('\n');
                }
            }
        }
    }
}
```

```

Method[] metodos = c.getDeclaredMethods();
Arrays.sort(metodos, Comparator.comparing(Method::getName));
for (Method m : metodos) {
    if (m.isSynthetic()) continue;
    List<String> params = new ArrayList<>();
    for (Class<?> p : m.getParameterTypes()) params.add(p.getSimpleName());
    sb.append(" ").append(visibilidad(m.getModifiers())).append(m.getName())
      .append('(').append(String.join(", ", params)).append(") : ")
      .append(m.getReturnType().getSimpleName()).append('\n');
}
sb.append("}\n");

if (c.getSuperclass() != null && clases.contains(c.getSuperclass())) {
    relaciones.add(c.getSuperclass().getSimpleName() + " <|-- " + nombre);
}
for (Class<?> i : c.getInterfaces()) {
    if (clases.contains(i)) relaciones.add(i.getSimpleName() + (c.isInterface() ? " <|-- " :
      " <|.. ") + nombre);
}
}
relaciones.forEach(r -> sb.append(r).append('\n'));
return sb.append("@enduml").toString();
}

public static void main(String[] args) {
    System.out.println(plantUml(MODELO));
}
}

```

La reflexión da toda la información del diagrama: `getDeclaredFields` y `getDeclaredMethods` (solo los de la propia clase, no los heredados), los modificadores de cada uno para la visibilidad, `getSuperclass` para la herencia y `getInterfaces` para la realización. Como el orden en que la JVM devuelve campos y métodos no está garantizado, se ordenan por nombre.

Lo interesante es decidir qué es un atributo y qué una asociación: si el tipo del campo es otra clase del modelo, es una relación con multiplicidad 1; si es una `List`, hay que mirar su tipo genérico con `getGenericType()` (el tipo «borrado» `List` no dice de qué es la lista) y, si es del modelo, es una asociación con multiplicidad *.

Las relaciones se guardan en una lista mientras se recorren las clases y se escriben al final, que es donde PlantUML las espera tras las definiciones.

13. Depura el programa de notas

Fácil · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

Solución (java)

```
import java.util.*;

public class Main {

    // Devuelve la media de las notas (con decimales)
    static double media(int[] notas) {
        int suma = 0;
        for (int i = 0; i < notas.length; i++) {      // error 1: empezaba en 1 y se saltaba la primera
            nota
            suma += notas[i];
        }
        return (double) suma / notas.length;          // error 2: la división entera perdía los decimales
    }

    // Devuelve la nota más alta
    static int maxima(int[] notas) {
        int max = notas[0];
        for (int n : notas) {
            if (n > max) max = n;                      // error 3: con < se quedaba con la mínima
        }
        return max;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] notas = new int[n];
        for (int i = 0; i < n; i++) notas[i] = sc.nextInt();
        System.out.printf(Locale.US, "Media: %.2f\n", media(notas));
        System.out.println("Máxima: " + maxima(notas));
    }
}
```

Los tres errores son clásicos: el bucle de la media empieza en 1 y se salta la primera nota; `suma / notas.length` es una división entre enteros y trunca el resultado; y en `maxima` la comparación está al revés, así que guarda la nota más baja.

Con el depurador se ven en seguida: al parar en el bucle, `i` empieza valiendo 1; al evaluar `suma / notas.length` en la vista de expresiones sale un número sin decimales; y al seguir `max` vuelta a vuelta, se ve que baja en lugar de subir.

14. Valores límite: clasificar edades

Fácil · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

Solución (java)

```
import java.util.*;

public class Main {

    static String clasificar(int edad) {
        if (edad < 0 || edad > 120) return "Error";
        if (edad <= 11) return "Niño";
        if (edad <= 17) return "Adolescente";
        if (edad <= 64) return "Adulto";
        return "Jubilado";
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) {
            System.out.println(clasificar(sc.nextInt()));
        }
    }
}
```

Primero se descartan los valores fuera de rango y después se comprueban los tramos de menor a mayor. Como cada `return` termina el método, cada condición solo necesita el límite superior de su tramo: si se llega a `edad <= 17`, ya se sabe que la edad es al menos 12.

Fíjate en que los límites se escriben tal como dice la especificación (`<= 11`, `<= 17`, `<= 64`): confundir un `<` con un `<=` es justo el error que detectan las pruebas de valores límite.

15. Caja negra: el problema del triángulo

Medio · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

Solución (java)

```
import java.util.*;

public class Main {

    static String tipo(int a, int b, int c) {
        if (a <= 0 || b <= 0 || c <= 0) return "No es triángulo";
        if (a >= b + c || b >= a + c || c >= a + b) return "No es triángulo"; // desigualdad triangular
        if (a == b && b == c) return "Equilátero";
        if (a == b || b == c || a == c) return "Isósceles";
        return "Escaleno";
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) {
            System.out.println(tipo(sc.nextInt(), sc.nextInt(), sc.nextInt()));
        }
    }
}
```

Primero se comprueba que los lados formen un triángulo: todos positivos y ninguno mayor o igual que la suma de los otros dos (con 1, 2 y 3 los lados quedarían planos). Hay que comprobar los tres lados, no solo el primero: por eso el caso «10 2 3» aparece en las pruebas con el lado largo en otra posición.

Después, el orden de las comprobaciones importa: equilátero antes que isósceles, porque un triángulo con los tres lados iguales también tiene dos iguales.

16. Refactoriza: extraer método

Medio · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

Solución (java)

```
import java.util.*;

public class Main {

    static final double IVA = 0.21;

    static double descuentoCliente(String tipo) {
        return switch (tipo) {
            case "VIP" -> 0.15;
            case "SOCIO" -> 0.05;
            default -> 0;
        };
    }

    static double descuentoVolumen(int unidades) {
        return unidades >= 10 ? 0.10 : 0;
    }

    static double aplicarDescuento(double importe, double descuento) {
        return importe * (1 - descuento);
    }

    static double precioFinal(double base, int unidades, String tipo) {
        double importe = base * unidades;
        importe = aplicarDescuento(importe, descuentoCliente(tipo));
        importe = aplicarDescuento(importe, descuentoVolumen(unidades));
        return importe * (1 + IVA);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in).useLocale(Locale.US);
        while (sc.hasNext()) {
            double base = sc.nextDouble();
            int unidades = sc.nextInt();
            String tipo = sc.next();
            System.out.printf(Locale.US, "%.2f%n", precioFinal(base, unidades, tipo));
        }
    }
}
```

Cada método extraído tiene un nombre que explica lo que hace y una sola responsabilidad, y los números mágicos han pasado a estar en un único sitio. `precioFinal` se lee ahora como la descripción del cálculo: importe, descuento del cliente, descuento por volumen e IVA.

Una refactorización no debe cambiar el comportamiento: los descuentos se aplican en el mismo orden y sobre el importe ya descontado, igual que en el código original. Por eso los resultados de las pruebas son idénticos.

17. Del diagrama de clases al código

Medio · Ejercicios de Entornos de Desarrollo · apuntesdam.com/subject/entornos/topic/ejercicios-entornos

Solución (java)

```
import java.util.*;

abstract class Figura {
    protected String nombre;

    public Figura(String nombre) { this.nombre = nombre; }

    public abstract double area();

    @Override
    public String toString() {
        return String.format(Locale.US, "%s: %.2f", nombre, area());
    }
}

class Circulo extends Figura {
    private double radio;

    public Circulo(double radio) {
        super("Círculo");
        this.radio = radio;
    }

    @Override
    public double area() {
        return Math.PI * radio * radio;
    }
}

class Rectangulo extends Figura {
    private double base;
    private double altura;

    public Rectangulo(double base, double altura) {
        super("Rectángulo");
        this.base = base;
        this.altura = altura;
    }

    @Override
    public double area() {
        return base * altura;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in).useLocale(Locale.US);
        List<Figura> figuras = new ArrayList<>();
        while (sc.hasNext()) {
            String tipo = sc.next();
            if (tipo.equals("C")) figuras.add(new Circulo(sc.nextDouble()));
            else figuras.add(new Rectangulo(sc.nextDouble(), sc.nextDouble()));
        }
        double total = 0;
        for (Figura f : figuras) {
            System.out.println(f);
            total += f.area();
        }
        System.out.printf(Locale.US, "Total: %.2f\n", total);
    }
}
```

Cada elemento del diagrama tiene su traducción directa: la flecha de herencia es `extends`, los atributos con «-» son campos `private` y el método abstracto de la clase padre obliga a cada subclase a dar su versión de `area()`.

El constructor de cada subclase llama primero a `super(...)` para inicializar la parte heredada (el nombre) y después guarda sus propios atributos. Gracias al polimorfismo, el `toString` de `Figura` llama a la `area()` de cada subclase sin saber cuál es.

18. Validador de DNI, NIE e IBAN diseñado con clases de equivalencia

Medio · Java · 50 minutos · apuntesdam.com/ejercicios/java/validador-dni-nie-iban

Solución (java)

```
import java.util.Scanner;

public class Main {
    static final String LETRAS = "TRWAGMYFPDXBNJZSQVHLCKE";

    /** null si el DNI es correcto, o el motivo por el que no lo es. */
    static String validarDni(String dni) {
        dni = dni.toUpperCase();
        if (!dni.matches("\\d{8}[A-Z]")) return "formato incorrecto (8 cifras y una letra)";
        char esperada = LETRAS.charAt(Integer.parseInt(dni.substring(0, 8)) % 23);
        return dni.charAt(8) == esperada ? null : "la letra debería ser " + esperada;
    }

    /** El NIE se valida como un DNI cambiando X, Y y Z por 0, 1 y 2. */
    static String validarNie(String nie) {
        nie = nie.toUpperCase();
        if (!nie.matches("[XYZ]\\d{7}[A-Z]")) return "formato incorrecto (X, Y o Z, 7 cifras y una letra)";
        return validarDni("XYZ".indexOf(nie.charAt(0)) + nie.substring(1));
    }

    /** IBAN español: 24 caracteres y control módulo 97 (ISO 13616). */
    static String validarIban(String iban) {
        iban = iban.replace(" ", "").toUpperCase();
        if (!iban.startsWith("ES")) return "solo se admiten IBAN españoles";
        if (iban.length() != 24) return "un IBAN español tiene 24 caracteres";
        if (!iban.substring(2).matches("\\d{22}")) return "después de ES solo puede haber cifras";
        // Se pasan los 4 primeros caracteres al final, las letras a números (A = 10 ... Z = 35)
        // y se calcula el resto entre 97 cifra a cifra, sin construir un número enorme
        String reordenado = iban.substring(4) + iban.substring(0, 4);
        int resto = 0;
        for (char c : reordenado.toCharArray()) {
            String valor = Character.isLetter(c) ? String.valueOf(c - 'A' + 10) : String.valueOf(c);
            for (char cifra : valor.toCharArray()) resto = (resto * 10 + (cifra - '0')) % 97;
        }
        return resto == 1 ? null : "los dígitos de control no cuadran";
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int total = 0, validos = 0;
        while (sc.hasNextLine()) {
            String linea = sc.nextLine().trim();
            if (linea.isEmpty()) continue;
            int espacio = linea.indexOf(' ');
            String tipo = (espacio < 0 ? linea : linea.substring(0, espacio)).toUpperCase();
            String valor = espacio < 0 ? "" : linea.substring(espacio + 1).trim();
            String error = switch (tipo) {
                case "DNI" -> validarDni(valor);
                case "NIE" -> validarNie(valor);
                case "IBAN" -> validarIban(valor);
                default -> "tipo de documento desconocido";
            };
            total++;
            if (error == null) validos++;
            System.out.println(tipo + (valor.isEmpty() ? "" : " : " + valor) + " → " + (error == null ? "válido" : "no válido: " + error));
        }
        System.out.println("Válidos: " + validos + " de " + total);
    }
}
```

Cada validador devuelve el motivo del error en lugar de un simple `boolean`. Eso hace el programa más útil para el usuario y las pruebas más precisas: no basta con saber que algo es inválido, hay que saber que lo es por la razón correcta.

El NIE se reduce a un DNI y se valida con el mismo método: reutilizar en lugar de copiar significa que un error corregido en el DNI queda corregido también en el NIE.

El cálculo del módulo 97 cifra a cifra se basa en que $(a \times 10 + b) \bmod 97 = ((a \bmod 97) \times 10 + b) \bmod 97$: nunca hay un número mayor que 970. Es lo que permite comprobar IBAN en cualquier lenguaje sin `BigInteger`.

Las pruebas salen de la tabla de clases de equivalencia y valores límite: un caso por cada grupo que el programa trata igual y los extremos (00000000 y 99999999). Con ocho casos bien elegidos se cubre lo que no cubrirían cien números al azar.

19. Depura el programa de notas: cinco errores escondidos

Medio · Java · 45 minutos · apuntesdam.com/ejercicios/java/depura-el-calculo-de-notas

Solución (java)

```
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Scanner;

public class Main {
    record Alumno(String nombre, double nota) {}

    /** Nota final: prácticas al 20 % cada una y examen al 60 %; con menos de un 4 en el examen, como
        mucho un 4. */
    static double notaFinal(double practica1, double practica2, double examen) {
        double nota = 0.2 * practica1 + 0.2 * practica2 + 0.6 * examen;
        if (examen < 4) nota = Math.min(nota, 4);
        return Math.round(nota * 10) / 10.0;
    }

    static String uno(double x) {
        return String.format(Locale.ROOT, "%.1f", x).replace('.', ',');
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<String> lineas = new ArrayList<>();
        while (sc.hasNextLine()) {
            String l = sc.nextLine().trim();
            if (!l.isEmpty()) lineas.add(l);
        }

        List<Alumno> alumnos = new ArrayList<>();
        for (int i = 0; i < lineas.size(); i++) {
            String[] p = lineas.get(i).split(";");
            alumnos.add(new Alumno(p[0], notaFinal(Double.parseDouble(p[1]), Double.parseDouble(p[2]),
                Double.parseDouble(p[3]))));
        }
        alumnos.sort((a, b) -> a.nota() != b.nota() ? Double.compare(b.nota(), a.nota()) :
            a.nombre().compareTo(b.nombre()));

        double suma = 0;
        int aprobados = 0;
        for (Alumno a : alumnos) {
            System.out.printf("%-20s%5s%n", a.nombre(), uno(a.nota()));
            suma += a.nota();
            if (a.nota() >= 5) aprobados++;
        }
        System.out.println("Media: " + uno(suma / alumnos.size()));
        System.out.println("Aprobados: " + aprobados + " de " + alumnos.size());
    }
}
```

Los cinco errores son clásicos: un índice que empieza en 1 y se salta al primer alumno; un copiar y pegar que suma dos veces la primera práctica; una comparación `<=` donde iba `<` (el examen de 4 sí compensa); una división entera que se come los decimales; y un `>` donde iba `>=` que deja fuera a quien saca un 5 justo.

Ninguno lo detecta el compilador ni produce una excepción: el programa da resultados plausibles. Solo se ven con casos de prueba que pasan por los bordes (un 4 en el examen, un 5 de nota final, el primer alumno de la lista) o calculando a mano un resultado.

El depurador ahorra tiempo cuando ya sabes dónde mirar: un punto de ruptura en `notaFinal` y la vista de variables muestran el valor de `nota` antes y después de cada línea, y ahí aparecen la práctica sumada dos veces y la división entera.

Cada error corregido debería convertirse en una prueba automática (una prueba de regresión), para que nadie lo vuelva a introducir. Es lo que hacen las pruebas ocultas de este ejercicio.