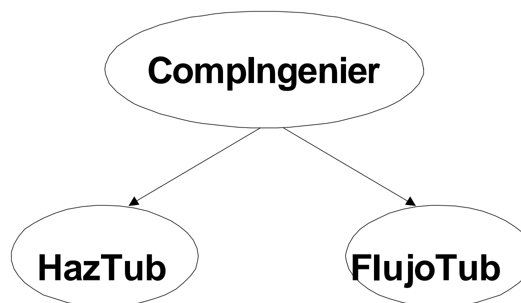
1.00 Clase 20

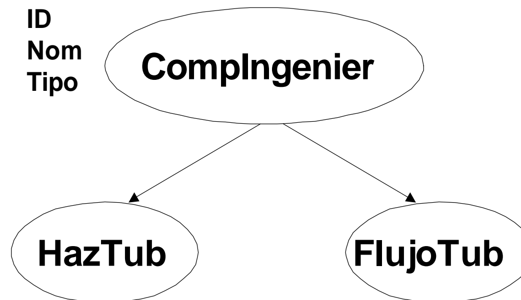
Resumen: Interfaces, cálculo de raíces
Integración, parte 1

Interfaces y herencia múltiple

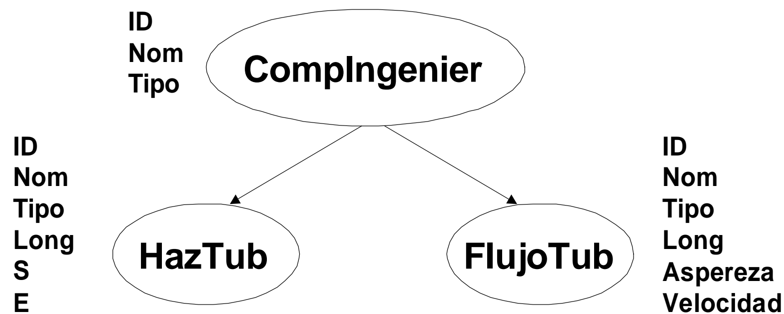


Supongamos un sistema de análisis de ingeniería que crea materiales, fluidos, etc. en varios componentes

Interfaces y herencia múltiple

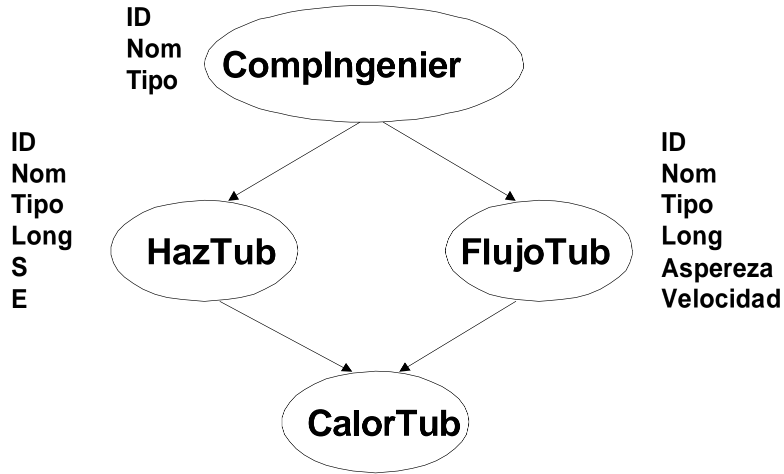


Interfaces y herencia múltiple



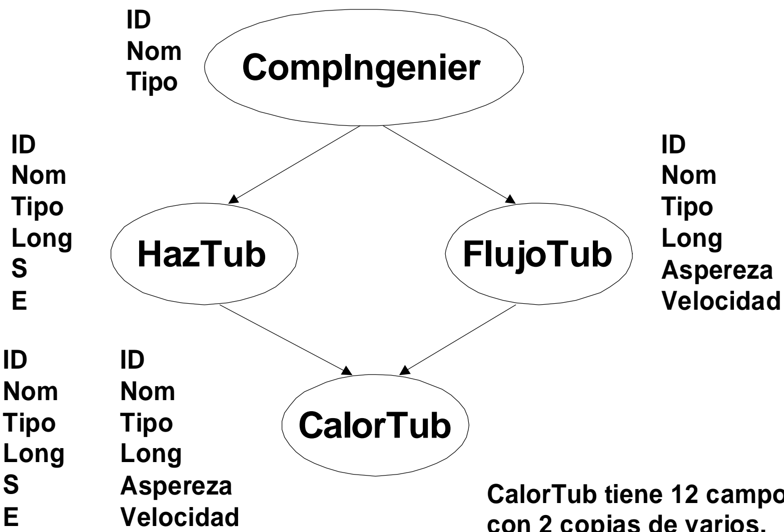
Ahora queremos un componente que conste de una tubería con flujo y haz circular. Queremos obtener su tasa de transferencia de calor a través del fluidos y el metals (p. ej. para enfriar una pista de hielo). Queremos *herencia múltiple* de HazTub y FlujoTub, los cuales disponen de métodos detrans ferencia de calor a través de fluidos y metales.

Interfaces y herencia múltiple



¿Qué campos tendrá CalorTub (en C++)?

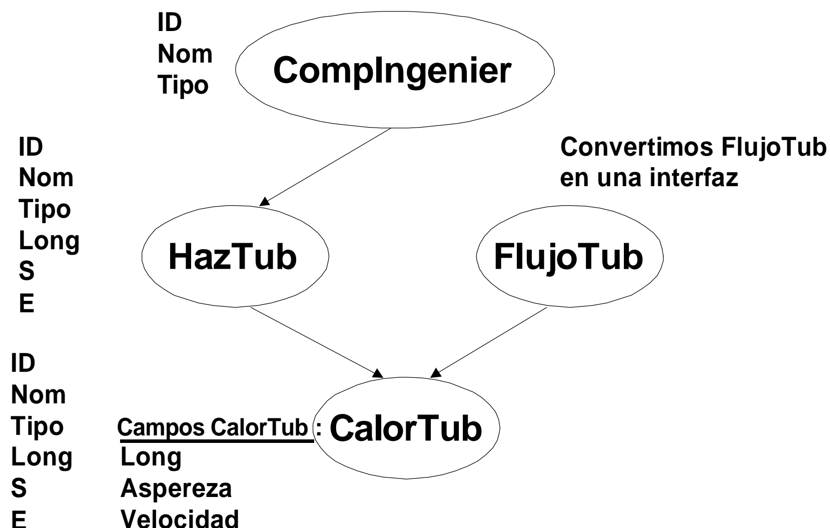
Interfaces y herencia múltiple



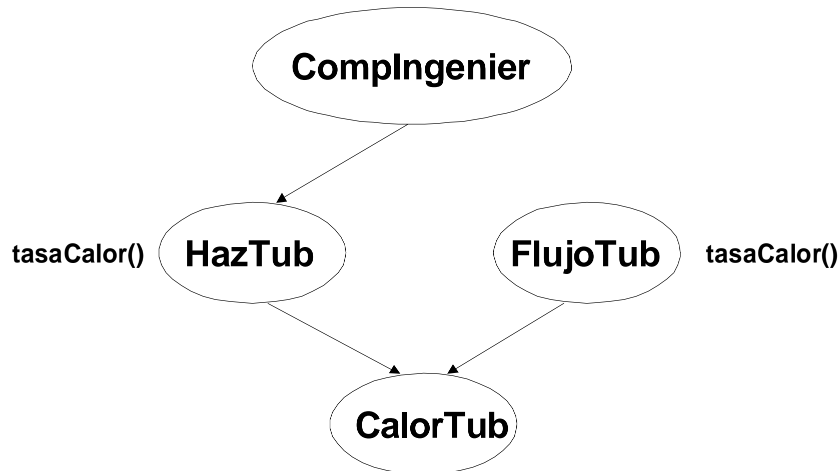
Interfaces y herencia múltiple

- Así, los campos en clases con herencia múltiple no son una buena idea:
  - Java implementa la herencia múltiple con interfaces, una clase abstracta muy restringida, para sortear esta dificultad
  - Java no permite campos de instancia en las interfaces, sólo los campos finales (constantes) que no implican esta dificultad
- Ahora, examinemos los métodos…
  - ¿Y si las interfaces permitiesen métodos no abstractos (métodos con cuerpos)?

Interfaces y herencia múltiple, parte 2



Interfaces y herencia múltiple, parte 2



¿Qué tasaCalor() es invocada por tasaCalor() o super.tasaCalor() en CalorTub?

Interfaces y herencia múltiple, parte 2

- Así, los métodos no abstractos en clases con herencia múltiple no son buena idea:
  - Java implementa la herencia múltiple con interfaces, una clase abstracta muy restringida, para sortear esta dificultad (sí, esto ya lo hemos dicho antes…)
  - Java no permite cuerpos de métodos en las interfaces, sólo métodos abstractos que no impliquen esta dificultad
  - Esto nos obliga a implementar el método en la subclase para que no exista ambigüedad al llamarlo
- Las interfaces son la forma que tiene Java de permitir que una clase tenga dos o más "tipos", esto es, que tenga herencia múltiple:
  - Las interfaces están muy restringidas en comparación con la herencia múltiple completa, pero son mucho más seguras y fáciles de entender. Y su utilidad justifica su uso, incluso si no ofrecen la reutilización que nos gustaría
  - Incluso si una interfaz se implementa dos veces por error en un conjunto de subclases, seguirá siendo totalmente clara (y esto ocurre)

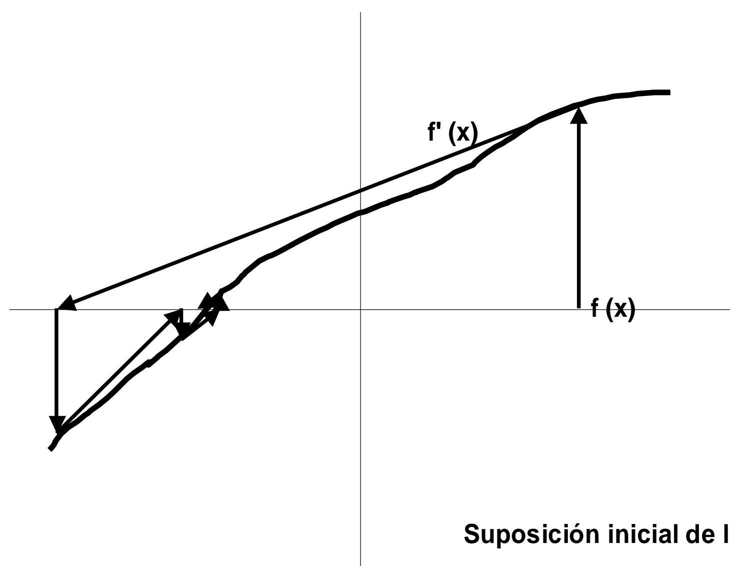
Método de Newton

- **Basado en la ampliación de la serie de Taylor:**

$$f(x+\delta) \approx f(x) + f'(x)\delta + f''(x)\delta^2/2 + \ldots$$

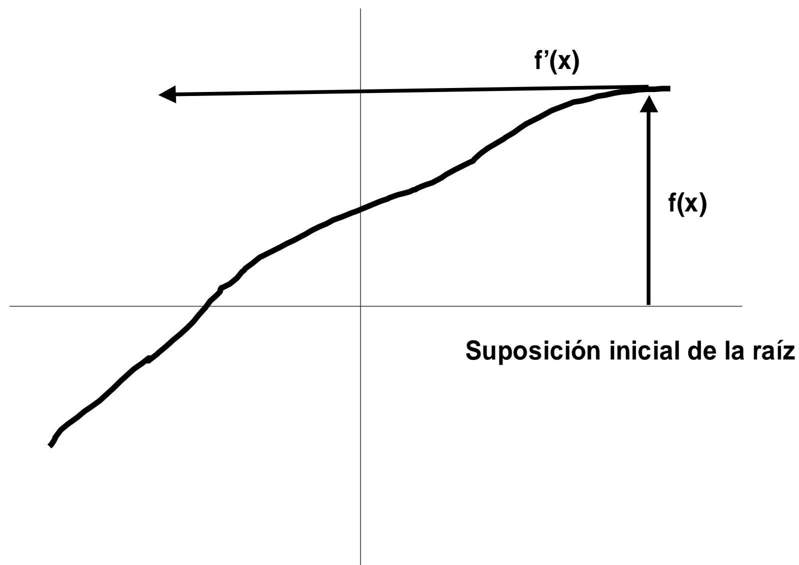
  - **Para incrementos pequeños y funciones suaves, las derivadas de orden superior son pequeñas y** $f(x+\delta) = 0$ **implica** $\delta = -f(x)/f'(x)$
  - **Si las derivadas de alto orden son grandes o la primera derivada es pequeña, Newton puede fallar**
  - **Converge rápidamente si se cumplen las premisas**
  - **Generalizable a N dimensiones (uno de los pocos)**
  - **Consulte la referencia de fórmulas para el método "seguro" de Newton- Raphson, que utiliza la bisección cuando la primera derivada es pequeña, etc.**

Método de Newton



Suposición inicial de la raíz

Patología del método de Newton



Método de Newton

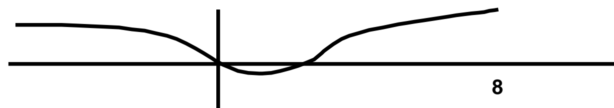
```
public class Newton {                        // RefForm, p. 365
    public static double newt(MathFunction2 func, double a,
                                    double b, double epsilon) {
        double guess= 0.5*(a + b);
        for (int j= 0; j < JMAX; j++) {
            double fval= func.fn(guess);
            double fder= func.fd(guess);
            double dx= fval/fder;
            guess -= dx;
            System.out.println(guess);
            if ((a - guess)*(guess - b) < 0.0) {
                System.out.println("Error: fuera de acotación");
                return ERR_VAL;        // Experimente con esto
            }                          // Es conservador
            if (Math.abs(dx) < epsilon)
                return guess;        }
        System.out.println("Superado el máximo de iteraciones");
        return guess;  }
```

Método de Newton, 2

```
    public static int JMAX= 50;
    public static double ERR_VAL= -10E10;

    public static void main(String[] args) {
        double root= Newton.newt(new FuncB(), -0.0, 8.0, 0.0001);
        System.out.println("Raíz: " + root);
        System.exit(0);     }  }

class FuncB implements MathFunction2 {
    public double fn(double x) {
        return x*x - 2;
    }
    public double fd(double x) {
        return 2*x;    }   }

public interface MathFunction2 {
    public double fn(double x);      // Valor de la función
    public double fd(double x);   }  // Valor de la primera derivada
```

Ejemplos

- $f(x)= x^2 + 1$
  - Sin raíces reales, Newton genera suposiciones aleatorias
- $f(x)= sen(5x) + x^2 – 3$ Raíz= -0.36667
  - Acotación entre –1 y 2, por ejemplo
  - La acotación entre 0 y 2 fallará con Newton conservador (fuera de la acotación)
- $f(x)= ln(x^2 –0.8x + 1)$ Raíces= 0, 0.8
  - La acotación entre 0 y 1.2 funciona
  - La acotación entre 0.0 y 8.0 falla

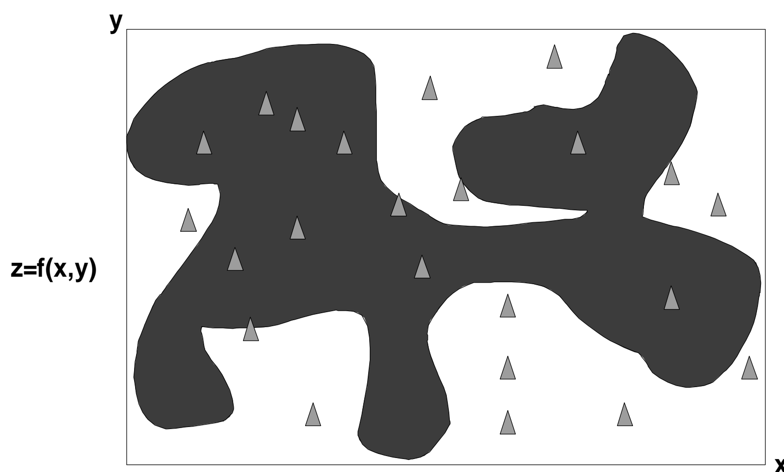


- Experimente con el archivo Roots.java de la práctica

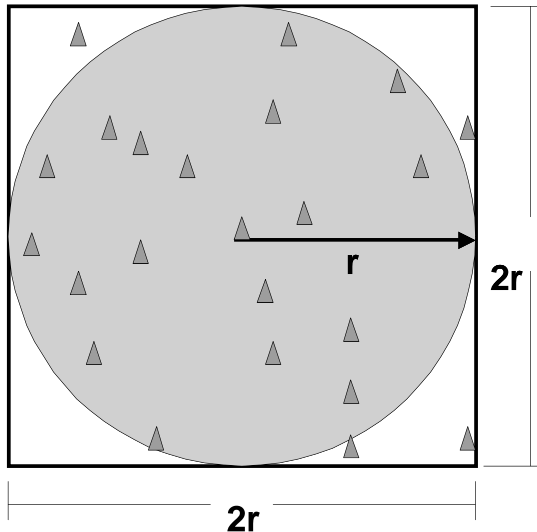
Integración numérica

- Los métodos clásicos sólo tienen interés histórico:
  - Rectangular, trapezoide, Simpson
  - Aptos para integrales muy suaves o que puedan calcularse analíticamente
- El método ampliado de Simpson: muy elemental; sólo de utilidad para integración 1D
- La integración multidimensional es difícil:
  - Si la región de la integración es compleja pero los valores de la función son suaves, use la integración Monte Carlo
  - Si la región es sencilla pero la función es irregular, divida la integración en regiones basadas en sitios conocidos de irregularidad
  - Si la región es compleja y la función es irregular, o si se deconocen los sitios de irregularidad de la función, déjelo estar
- Abordaremos sólo el método ampliado de Simpson 1D
  - Para más información consulte la ref. de fórmulas (capítulo 4)

Integración Monte Carlo



Cálculo de Pi



Genere aleatoriamente puntos en el cuadrado $4r^2$. La probabilidad de que estén en el círculo es $\pi r^2 / 4r^2$, o $\pi / 4$.

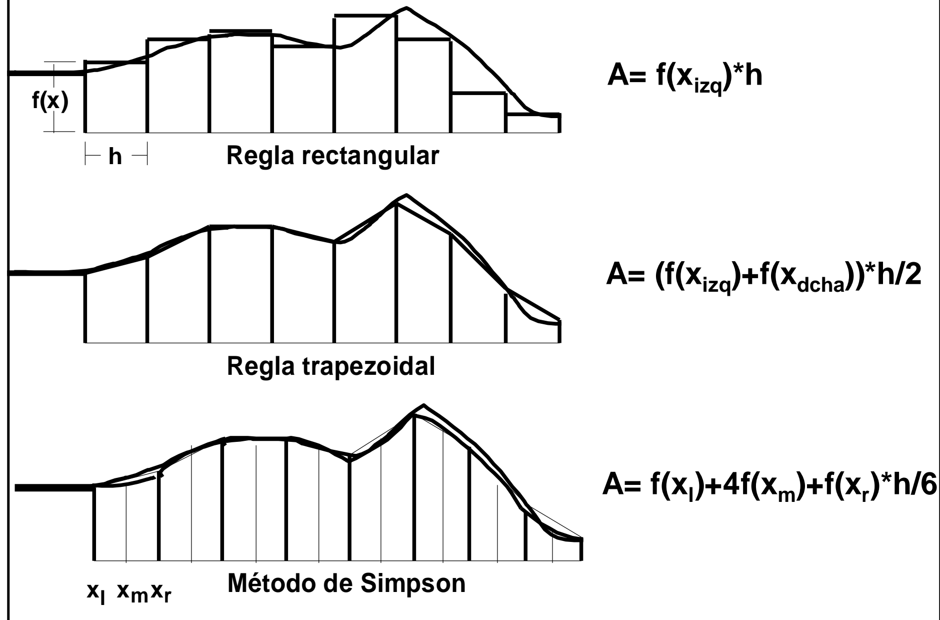
Esta es la integración Monte Carlo con f(x,y)= 1

Si f(x,y) varía despacio, calcule f(x,y) en cada punto de muestra de los límites de integración y suma

CalcularPi (GetPi)

```
public class GetPi {
    public static double getPi() {
        int count=0;
        for (int i=0; i < 1000000; i++) {
            double x= Math.random() - 0.5;  // Ctr at 0,0
            double y= Math.random() - 0.5;
            if ((x*x + y*y) < 0.25)    // Si está en la región
                count++;          // incrementa la integral
        }                         // Más general, eval f()
        return 4.0*count/1000000.0;         // Valor de la integral
    }

    public static void main(String[] args) {
        System.out.println(getPi());
        System.exit(0);
    }
}
```

Métodos elementales



Métodos elementales

```
class FuncA implements MathFunction {
    public double f(double x) {
        return x*x*x*x +2;    }   }

public class Integration {
    public static double rect(MathFunction func,
                     double a, double b, int n) {
        double h= (b-a)/n;
        double answer=0.0;
        for (int i=0; i < n; i++)
            answer += func.f(a+i*h);
        return h*answer;    }

    public static double trap(MathFunction func,
                     double a, double b, int n) {
        double h= (b-a)/n;
        double answer= func.f(a)/2.0;
        for (int i=1; i <= n; i++)
            answer += func.f(a+i*h);
        answer -= func.f(b)/2.0;
        return h*answer;    }
```

Métodos elementales, 2

```
 public static double simp(MathFunction func,
                    double a, double b, int n) {
       // Cada panel tiene un área de (h/6)*(f(x) + 4f(x+h/2) + f(x+h))
       double h= (b-a)/n;
       double answer= func.f(a);
       for (int i=1; i <= n; i++)
           answer += 4.0*func.f(a+i*h-h/2.0)+ 2.0*func.f(a+i*h);
       answer -= func.f(b);
       return h*answer/6.0;    }

   public static void main(String[] args) {
       double r= Integration.rect(new FuncA(), 0.0, 8.0, 200);
       System.out.println("Rectángulo: " + r);
       double t= Integration.trap(new FuncA(), 0.0, 8.0, 200);
       System.out.println("Trapezoide: " + t);
       double s= Integration.simp(new FuncA(), 0.0, 8.0, 200);
       System.out.println("Simpson: " + s);
       System.exit(0);
   }
}  //Problemas: el cálculo no es preciso, ineficaz, enteros cerrados
```