NOTAS DE LA CLASE CUENTA:

1 -

```
private double saldo = 0;
private List<Movimiento> movimientos = new ArrayList<>();

public Cuenta() {
  saldo = 0;
}

public Cuenta(double montoInicial) {
  saldo = montoInicial;
}
```

Es innecesario tener por defecto el saldo en 0 cuando se está cubriendo las inicializaciones de los parámetros en ambos constructores.
Si bien de esta forma se ve más explícitamente las intenciones de iniciar el saldo en 0, lo decidí sacar por el hecho que es redundante.

2-

```
public void poner(double cuanto) {
  if (cuanto <= 0) {
    throw new MontoNegativoException(cuanto + ": el monto a ingresar debe ser un valor positivo");
  }

  if (getMovimientos().stream().filter(movimiento -> movimiento.isDeposito()).count() >= 3) {
    throw new MaximaCantidadDepositosException("Ya excedio los " + 3 + " depositos diarios");
  }

  new Movimiento(LocalDate.now(), cuanto, true).agregateA(this);
}
```

En el segundo if de la imagen surgen dos temas.

```
if (getMovimientos().stream().filter(movimiento -> movimiento.isDeposito()).count() >= 3) {
  throw new MaximaCantidadDepositosException("Ya excedio los " + 3 + " depositos diarios");
}
```

El primero es el hecho que la excepción no es coherente con lo que se está validando. Ya que en la validación se verifica si la cantidad de movimientos que tiene la cuenta es mayor a 3, pero no se está verificando la fecha del movimiento para saber si fueron realizados el mismo día.
El segundo va por por el lado de *"messages chain"*. Es un mensaje largo que se puede abstraer su lógica para ser reutilizado.

Por otra parte, el método ‘poner’ es el encargado de crear un nuevo movimiento y agregarlo a su lista de movimientos.

```
new Movimiento(LocalDate.now(), cuanto, true).agregateA(this);
```

Para realizar esto, le envía un mensaje a *Movimiento* pasandose como parámetro para hacer un pasamano de lógica. Lo cual se podría simplificar creando la instancia y añadiendola a su lista de movimientos.

3-

```
public void sacar(double cuanto) {
  if (cuanto <= 0) {
    throw new MontoNegativoException(cuanto + ": el monto a ingresar debe ser un valor positivo");
  }
  if (getSaldo() - cuanto < 0) {
    throw new SaldoMenorException("No puede sacar mas de " + getSaldo() + " $");
  }
  double montoExtraidoHoy = getMontoExtraidoA(LocalDate.now());
  double limite = 1000 - montoExtraidoHoy;
  if (cuanto > limite) {
    throw new MaximoExtraccionDiarioException("No puede extraer mas de $ " + 1000
        + " diarios, límite: " + limite);
  }
  new Movimiento(LocalDate.now(), cuanto, false).agregateA(this);
}
```

No seria muy recomendable juntar esas validaciones en un lugar haciendo algo como:
if (cuanto <= 0 && getSaldo() - cuanto < 0){ ... }
De la forma que está originalmente el usuario puede saber con más detalle cuál fue el problema a comparación de tirar un solo error con el mensaje *“cantidad inválida”,* además de hacer el código más robusto por si tenemos que manejar esos errores de alguna forma.
Sin embargo esas dos verifica ciones las podemos sacar afuera y juntarlas en un solo método para hacer el método más corto y entendible.
Por otra parte, al igual que el método *‘poner’* , también es un *Long Method* que podría ser dividido delegando su lógica.

4-
Mas abajo, se llama al metodo ‘*getMontoExtraidoA’*

```
public double getMontoExtraidoA(LocalDate fecha) {
  return getMovimientos().stream()
      .filter(movimiento -> !movimiento.isDeposito() && movimiento.getFecha().equals(fecha))
      .mapToDouble(Movimiento::getMonto)
      .sum();
}
```

que se podría mejorar separando la lógica en dos partes:
filtrar las extracciones en una fecha y sumar esas extracciones.

5-
El siguiente método está de más si es llamado desde la misma clase. Porque solo sirve como un *'pasa manos'* de un atributo que ya le corresponde a la clase. Por lo que es innecesario.

```
public List<Movimiento> getMovimientos() {
  return movimientos;
}
```

Si el método estuviese y fuese usado de la misma forma en una ***subclase*** de Cuenta, entonces estaría bien, ya que evitaría el code smell 'inappropriate intimacy'. Pero este no es el caso.
Es decir, el método no está mal que exista. Ya que es un getter y puede servir para los test, pero está mal la forma en la que se lo usa dentro de la misma clase.

NOTAS DE LA CLASE MOVIMIENTO

1-
Una de las primeras cosas que se ve en la clase *Movimiento* es que posee un booleano 'esDeposito'.

```
private boolean esDeposito;
```

Ese atributo está presente en varios métodos que tienen comportamiento similar pero difieren en el valor de este booleano, por ejemplo:

```
public boolean fueDepositado(LocalDate fecha) {
  return isDeposito() && esDeLaFecha(fecha);
}

public boolean fueExtraido(LocalDate fecha) {
  return isExtraccion() && esDeLaFecha(fecha);
}
```

Siendo los métodos '*isDeposito'* y *'isExtraccion':*

```
public boolean isDeposito() {
  return esDeposito;
}

public boolean isExtraccion() {
  return !esDeposito;
}
```

También a la hora de tomar alguna decisión, está presente este booleano para saber por qué camino debe ir:

```
public double calcularValor(Cuenta cuenta) {
  if (esDeposito) {
    return cuenta.getSaldo() + getMonto();
  } else {
    return cuenta.getSaldo() - getMonto();
  }
}
```

Todos estos comportamientos mencionados resaltan que la clase está dividida en dos ramas con alguna lógica y atributos en común.
Por lo tanto, esto se puede resolver utilizando polimorfismo. Si ya se tiene en claro que existen dos tipos de movimientos, se puede tratar a las mismas como subclases de Movimiento y que el comportamiento cambie en cada una de ellas.

2-

```
public void agregateA(Cuenta cuenta) {
  cuenta.setSaldo(calcularValor(cuenta));
  cuenta.agregarMovimiento(fecha, monto, esDeposito);
}
```

Este método es llamado desde la clase Cuenta para agregarse un movimiento a su lista de movimientos.
Como se dijo antes, esto está mal porque cuenta se pasa a si mismo por parametro hacia este método y este método lo único que hace es modificar sus parámetros. Así que además de ser innecesario, está rompiendo el encapsulamiento de cuenta, porque se está metiendo en sus atributos y modificandolos.

NOTAS DE LOS TESTS

1-

```
@Test
void TresDepositos() {
  cuenta.poner(1500);
  cuenta.poner(456);
  cuenta.poner(1900);
}
```

Como este, hay varios ejemplos en los cuales los tests no tienen assertions, lo cual esta mal. Los tests son para probar el correcto comportamiento de una clase, no para realizar instancias.

NOTAS GENERALES

Hablando en general, existen casos en los cuales los métodos tienen poca expresividad. No reflejan muy bien su propósito. Por ejemplo, en la clase Cuenta, existe el método 'poner', que en ese contexto puede hacer referencia al saldo como también al movimiento. Lo mismo con 'sacar'.
Esto se nota más en los tests, que a simple vista no se sabe que se está testeando. Se puede deducir recién viendo el código del test.

CAMBIANDO EL DISEÑO

Me hacía un poco de ruido que toda la lógica de verificar si una cuenta está en condiciones de realizar un depósito/extracción esté en la clase *Cuenta*. Tampoco me parecía bien el hecho que la clase Cuenta agregue Validaciones a sus respectivas posibles operaciones. Por un tema de polimorfismo y flexibilidad mejor que cada tipo de operación sepa asignar sus propias validaciones.
Por estos motivos decidí mover toda la lógica de asignación y evaluación de *Validaciones* a sus respectivas operaciones. De esta forma aprovecho el polimorfismo y me deshago de esa lógica que no me interesa que esté en la clase Cuenta.

La clase Cuenta quedó así:

```
public void realizarDeposito(double monto){
  new Deposito().evaluarConcrecion(monto, cuenta: this);
}

public void realizarExtraccion(double monto){
  new Extraccion().evaluarConcrecion(monto, cuenta: this);
}
```

El problema que le veo es que cada vez que la cuenta reciba el mensaje para realizar una operación, se va a crear una instancia de Extracción/Depósito. En wollok podría realizarse de otra forma porque se podrían tratar como WKO, pero acá desconozco si existe algo parecido.
Me fije cual era la mejor forma de implementar esto. Si era con enums, singletons de Extracción/Depósito o haciendo los métodos static. En todos tuve problemas.
En los enums no podía poner los métodos de la forma que quería, y también me parecía que estaba de más.
Los singletons no me parecieron buena idea, ya que quiero tener esas entidades en la lista de movimientos de la cuenta.
Y con los métodos static tuve el problema que tenia que hacer todos los métodos/atributos que están relacionados de tipo static. Por lo que no podía abstraer cierta lógica en la clase Movimiento. Por lo que la terminé descartando.

Validaciones:

Modifique las validaciones para que sean clases que implementen la interface Validación.

```
public interface Validacion {

  public void validar(Cuenta cuenta, double monto);

}
```

Me pareció mejor tenerlos de esta forma por la facilidad de crear nuevas validaciones y agregarlas a los diferentes tipos de operaciones. Y ya que todos implementan y saben responder al mensaje *'Validar()'* puedo aprovechar el polimorfismo haciendo algo como:

```
public void evaluarValidaciones(Cuenta cuenta, double monto){
  validaciones.forEach(v -> v.validar(cuenta, monto));
}
```