¿De dónde vienen?

- Explicación Pragmática
- Explicación Conceptual

Explicación Pragmática

Código “Robusto” por medio de técnica de

Código de Retorno

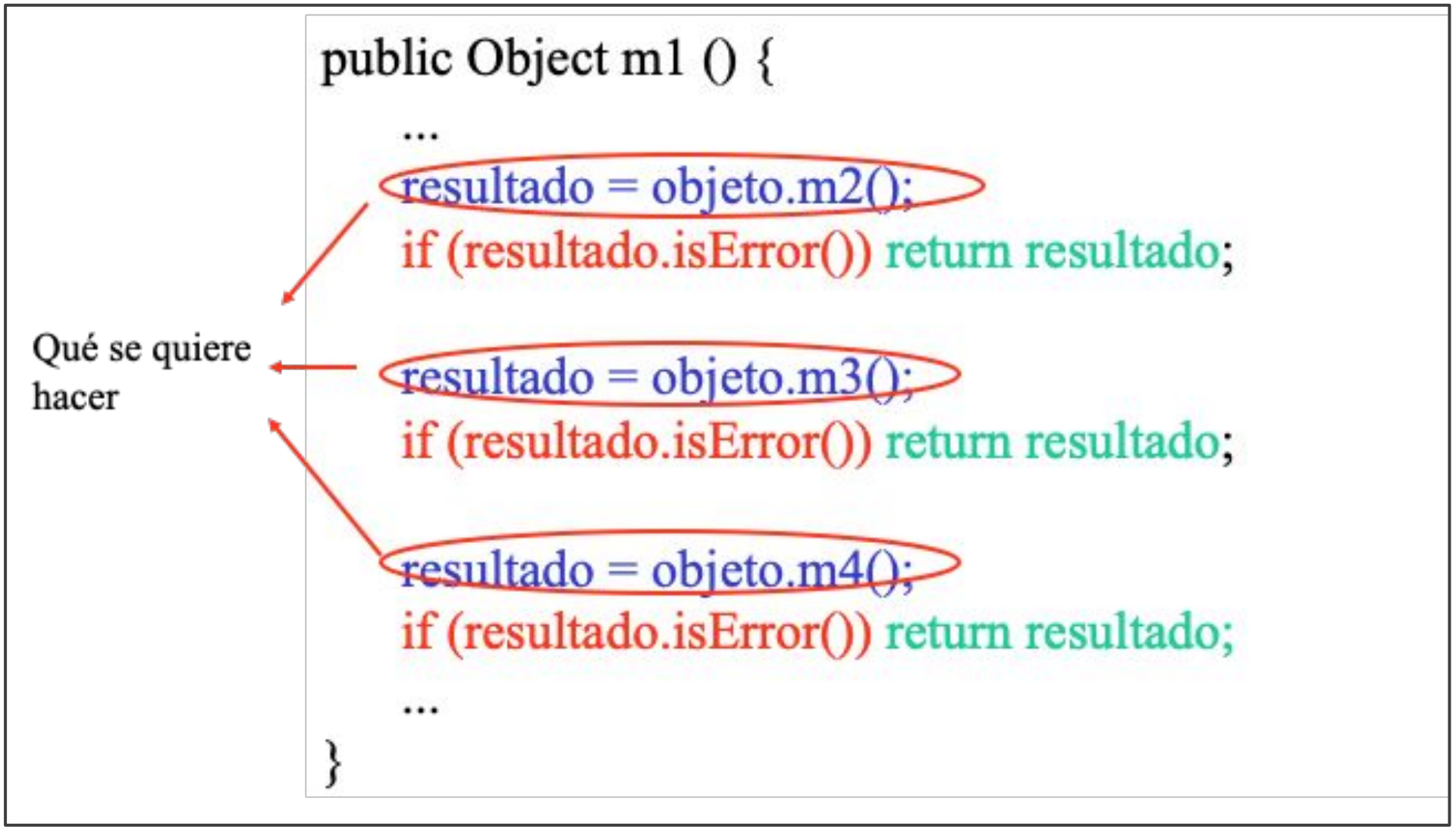
Ejemplo - Técnica de Código de Retorno

```
public Object m1 () {
        ...
        resultado = objeto.m2();
        if (resultado.isError()) return resultado;

        resultado = objeto.m3();
        if (resultado.isError()) return resultado;

        resultado = objeto.m4();
        if (resultado.isError()) return resultado;
        ...
}
```

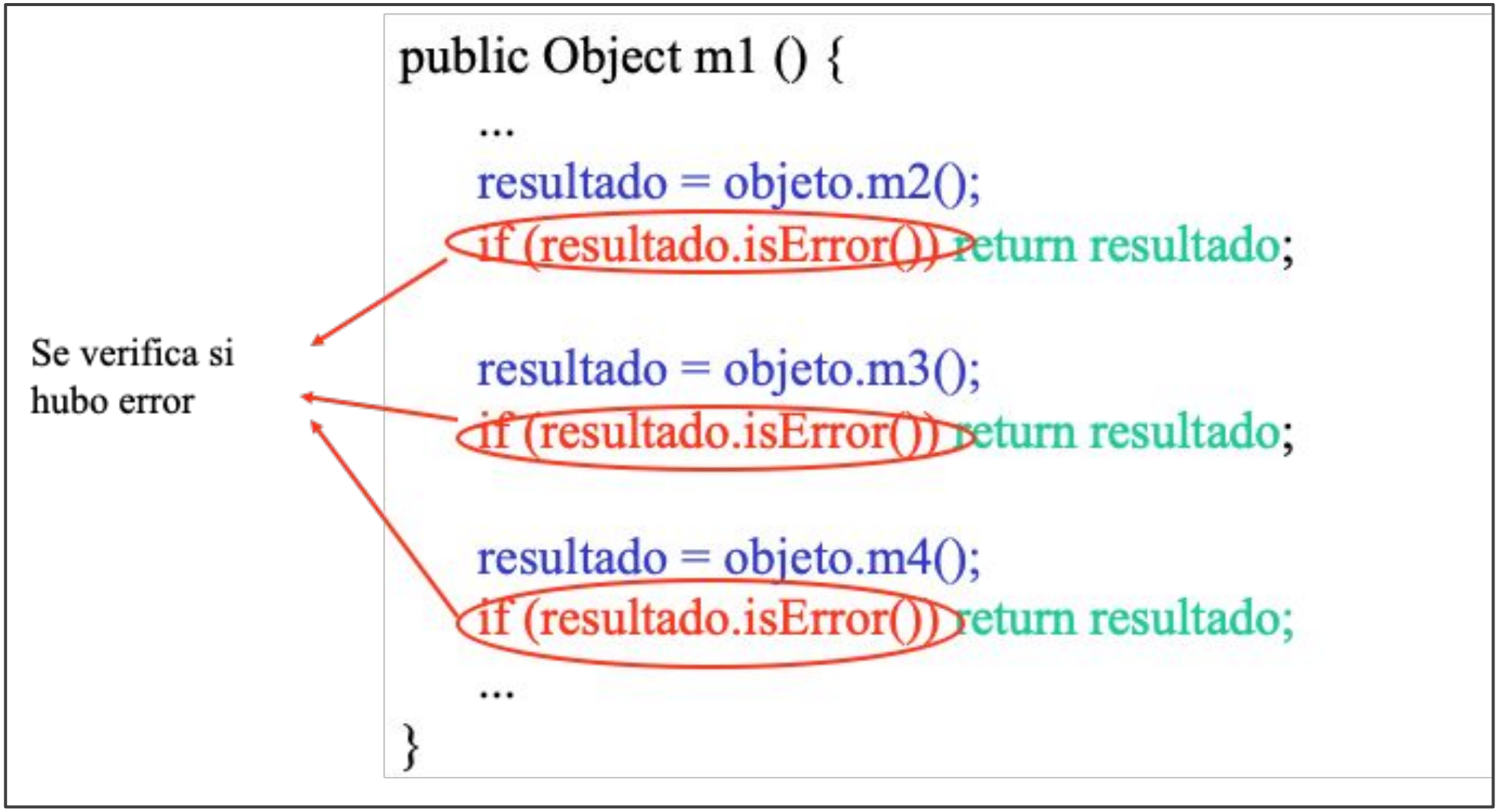
Problemas

- **Código Repetido** → Problema de diseño
- **Propenso a error** → Nos podemos olvidar de poner el if isError...
- **Difícil de leer qué** se quiere hacer por estar entremezclado el código de adm. de error
- **No está estandarizado** (salvo en Go)
  - ¿Qué pasa si se quiere devolver algo más allá del error?
  - ¿Cómo se resuelve a nivel tipos?

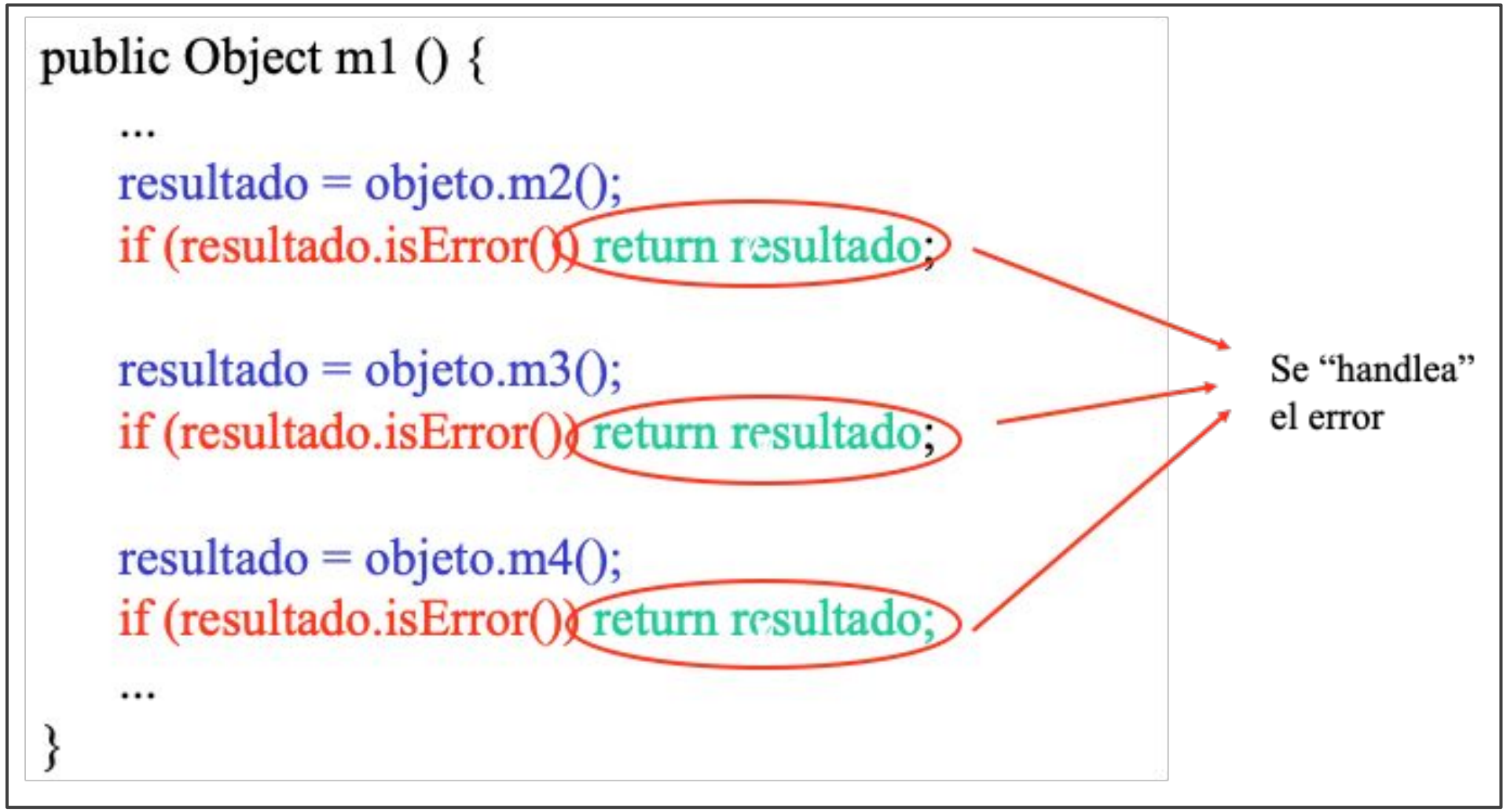
Técnica de Código de Retorno



Técnica de Código de Retorno



Técnica de Código de Retorno



Explicación Pragmática

Las excepciones son el resultado de sacar código repetido de la técnica de **“código de retorno”**

Sacar Código Repetido de la Técnica de Código de Retorno

```
public Object m1 () {
    ...
    resultado = objeto.m2();
    if (resultado.isError()) return resultado;

    resultado = objeto.m3();
    if (resultado.isError()) return resultado;

    resultado = objeto.m4();
    if (resultado.isError()) return resultado;
    ...
}
```

```
public Object m1 () {
    INTENTAR {
    objeto.m2();
    objeto.m3();
    objeto.m4();
    } isError() {
      return resultado;
    }
}
```

Excepciones en lenguajes con sintaxis tipo C

```
try {
    doSomething();
} catch (RuntimeException e) {
    handle(e);
}
```

Excepciones en lenguajes con sintaxis tipo C

```
try {
    doSomething();
} catch (RuntimeException e) {
    handle(e);
}
```

Condición de Handleo.
Acoplamiento con tipo de Excepción

Excepciones con Objetos y Mensajes

```
[ self doSomething ]
        on: Error
        do: [ :anError | self handle: anError ]
```

Excepciones con Objetos y Mensajes

```
try {
    doSomething();
} catch (RuntimeException e) {
    handle(e);
}
```

```
[ self doSomething ]
    on: Error
    do: [ :anError | self handle: anError ]
```

Excepciones con Objetos y Mensajes

```
try {
    doSomething();
} catch (RuntimeException e) {
    handle(e);
}
```

```
[ self doSomething
    on: Error
    do: [ :anError | self handle: anError ]
```

Excepciones con Objetos y Mensajes

```
try {
    doSomething();
} catch (RuntimeException e) {
    handle(e);
}
```

```
[ self doSomething ]
    on: Error
    do: [ :anError | self handle: anError ]
```

Explicación Conceptual

- Programación por Contratos - Bertrand Meyer
- Contratos explícitos
- Contratos implícitos
- ¿Quién verifica los contratos?
- ¿Qué hacer cuando no se cumple un contrato?

https://wiki.c2.com/?DesignByContract

Explicación Conceptual

Basada en la técnica de **Design by Contract**

(Bertrand Meyer - https://wiki.c2.com/?DesignByContract)

¿Qué es un Contrato?

Acuerdo de voluntades entre partes que genera derechos y obligaciones y a cuyo cumplimiento pueden ser compelidas.

Contrato

- Características:
  - Orales o Escritos
  - Contratos explícitos o Implícitos
  - Legales
- Acciones relacionadas:
  - ¿Quién verifica los contratos?
  - ¿Qué hacer cuando no se cumple un contrato?

¿Qué es un Contrato en Objetos?

Condiciones que se deben cumplir entre los objetos que colaboran y que de hacerlo, aseguran un resultado definido algorítmicamente

Condiciones de Contratos en Objetos

- Pre-Condiciones
- Post-Condiciones
- Invariantes

Pre-Condiciones

- Condiciones que se deben mantener para poder ejecutar un método
  - Ej: Monto a extraer de una caja de ahorro debe ser >= 0
- ¿Quién debe verificarlas? → más adelante

Post-Condiciones

- Condiciones que se deben mantener después de ejecutar un método
  - Ej. en una extracción:
    *saldo = **prev**(saldo)-montoAExtraer*
  - Ej. en un algoritmo de Sorting:
    La colección está ordenada
- Cuando se verifican:
  - Soporte explícito del lenguaje (Ej. Eiffel)
  - Tests

Invariantes

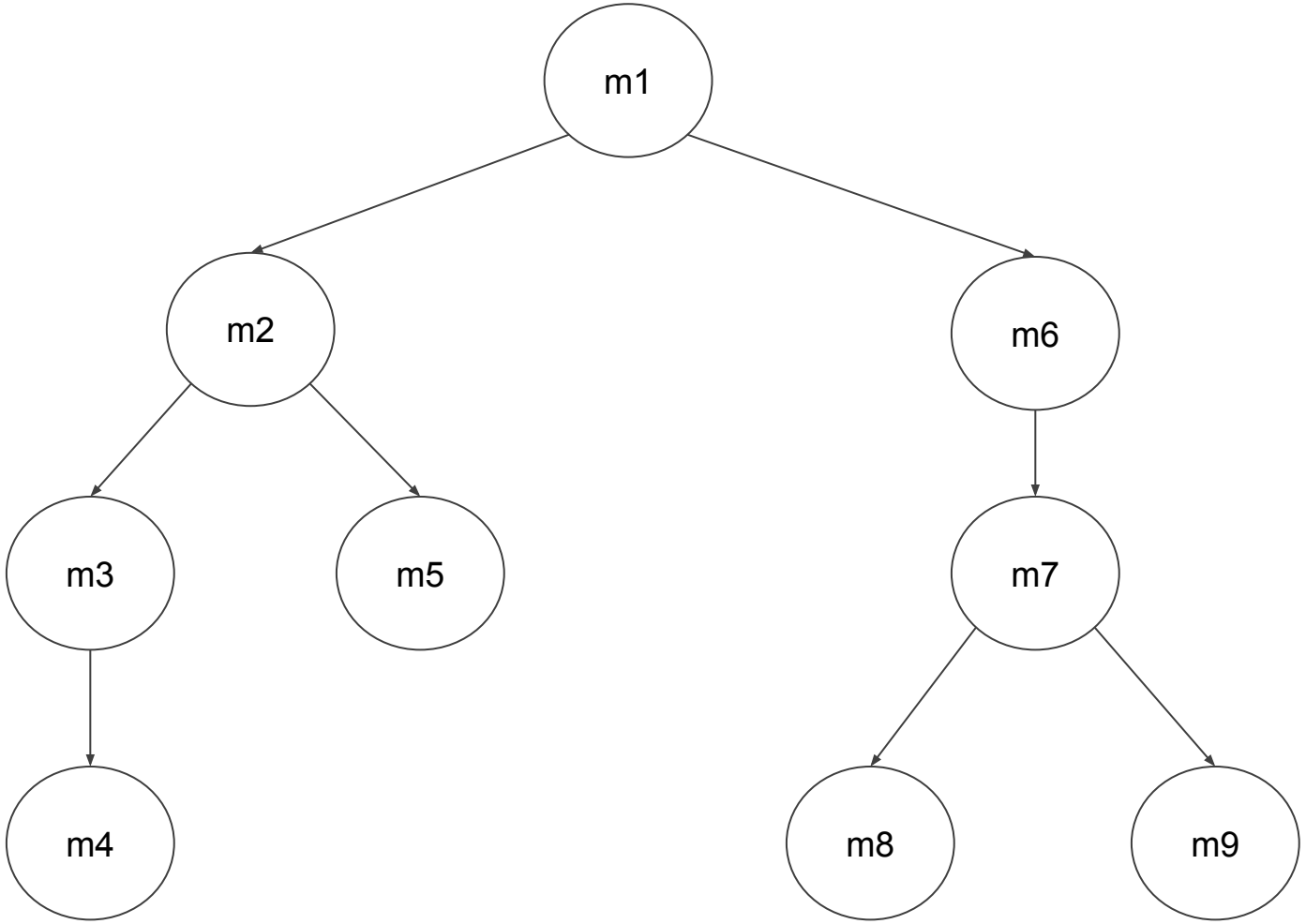
- Condiciones que siempre se deben mantener en las instancias de una clase
  - Ej: El saldo de una cuenta bancaria siempre debe ser >= 0
- Cuándo se verifican:
  - Soporte explícito del lenguaje (Ej. Eiffel)
  - Tests

Explicación Conceptual

Las excepciones se utilizan para indicar que **no se cumple un contrato**

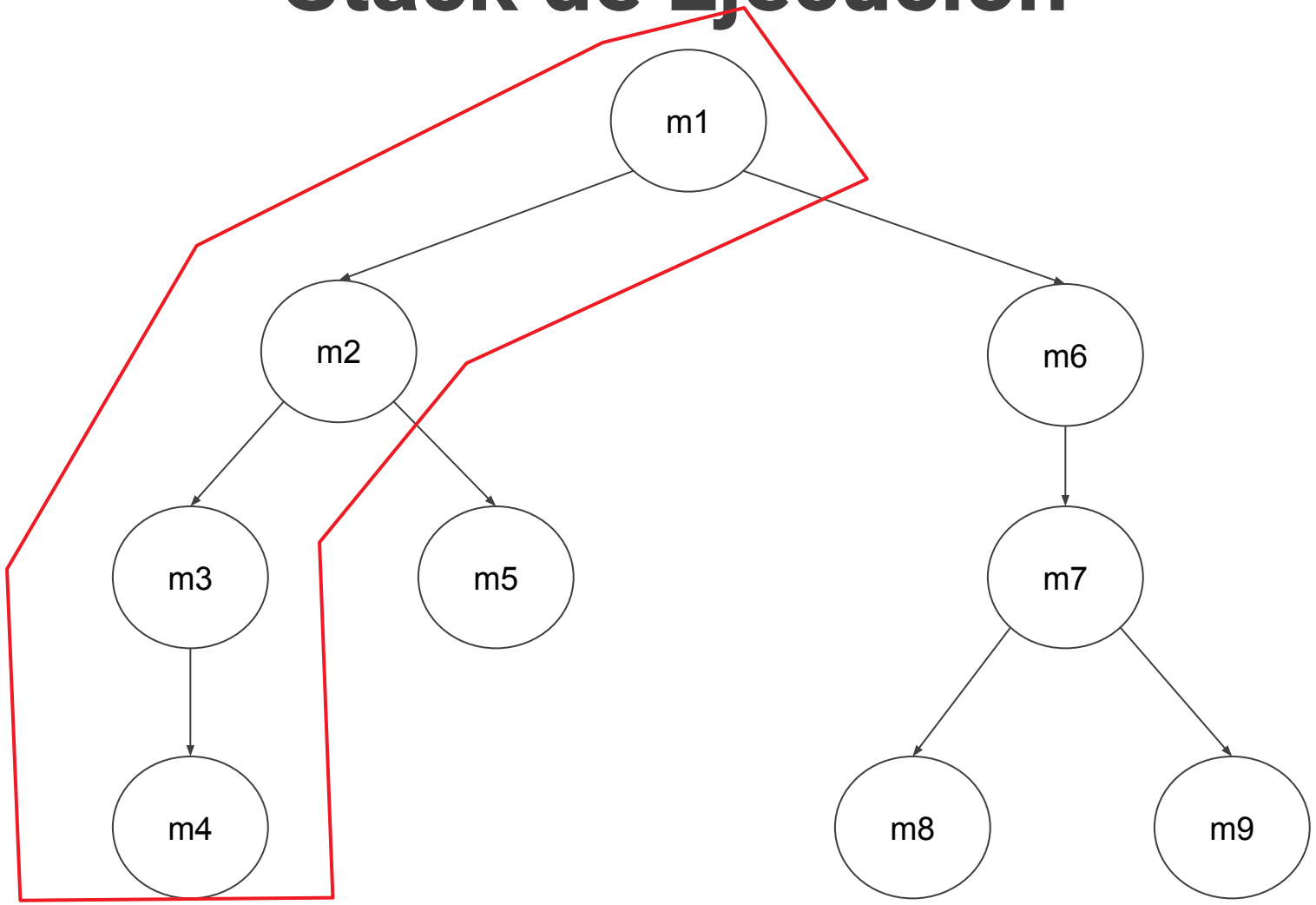
¿Cómo funcionan?

Árbol de Ejecución



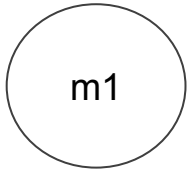
¿Cómo funcionan?

Stack de Ejecución



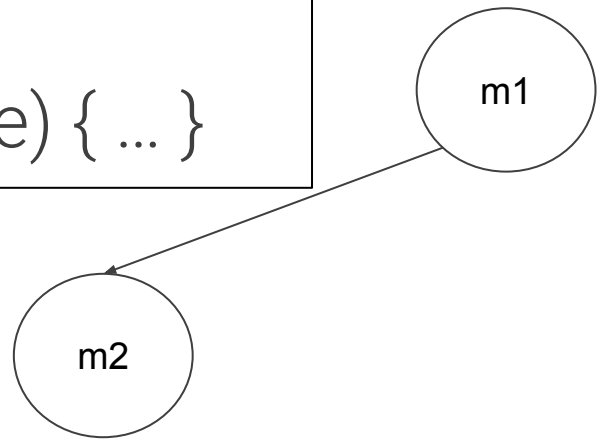
¿Cómo funcionan?

```
try { self m2 }
catch (RuntimeError e) { … }
```



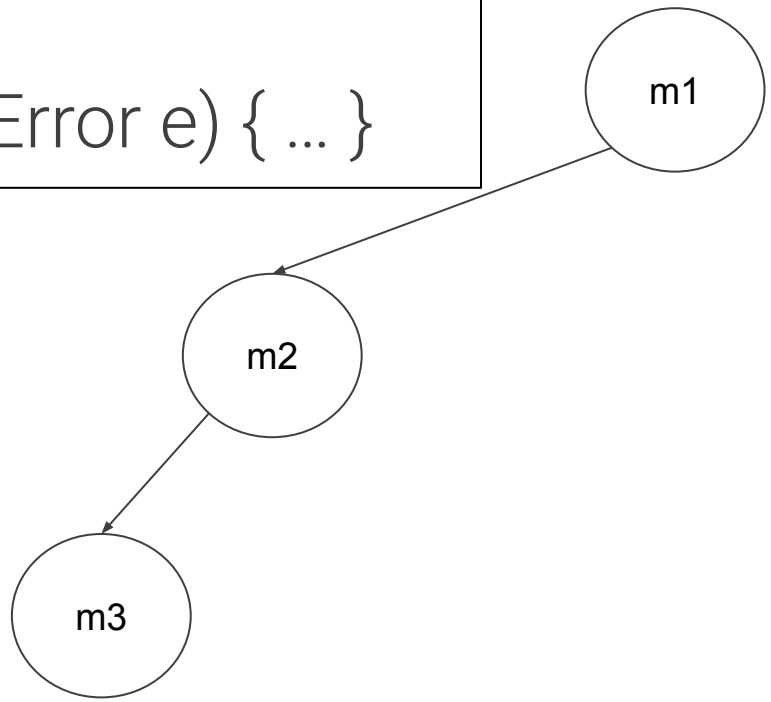
¿Cómo funcionan?

```
try { self m2 }
catch (RuntimeError e) { ... }
```



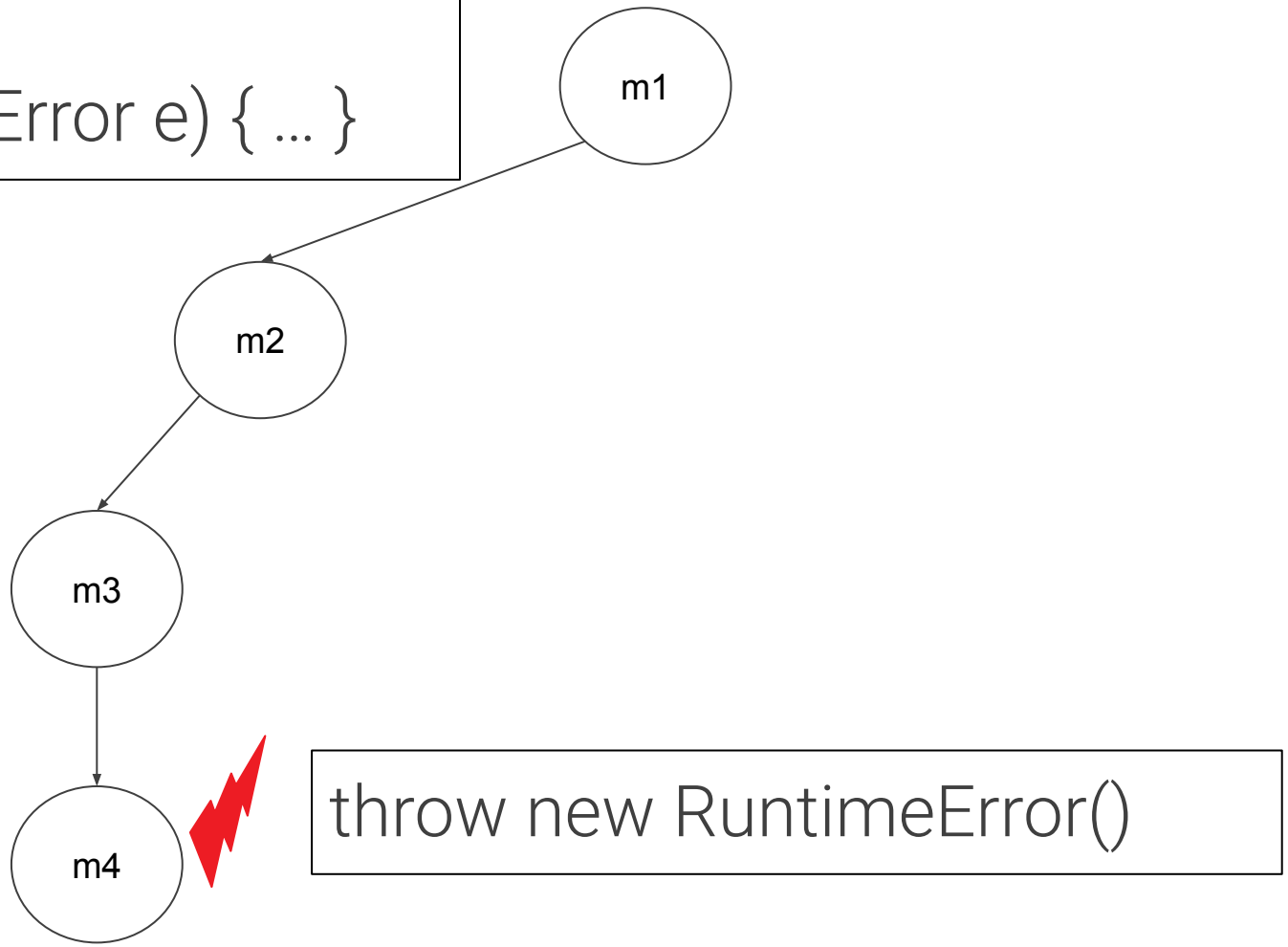
¿Cómo funcionan?

```
try { self m2 }
catch (RuntimeError e) { ... }
```



¿Cómo funcionan?

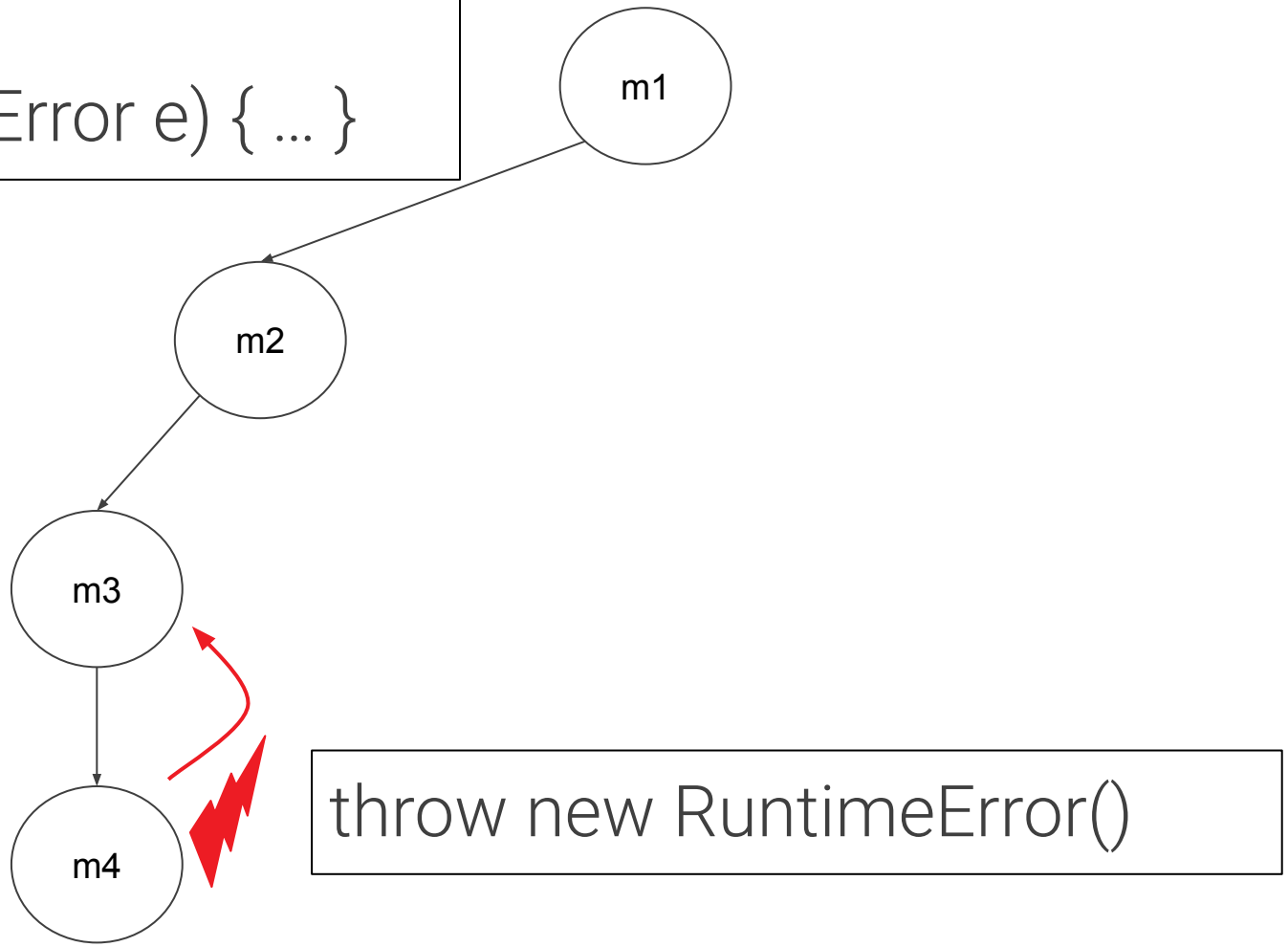
```
try { self m2 }
catch (RuntimeError e) { ... }
```




```
throw new RuntimeError()
```

¿Cómo funcionan?

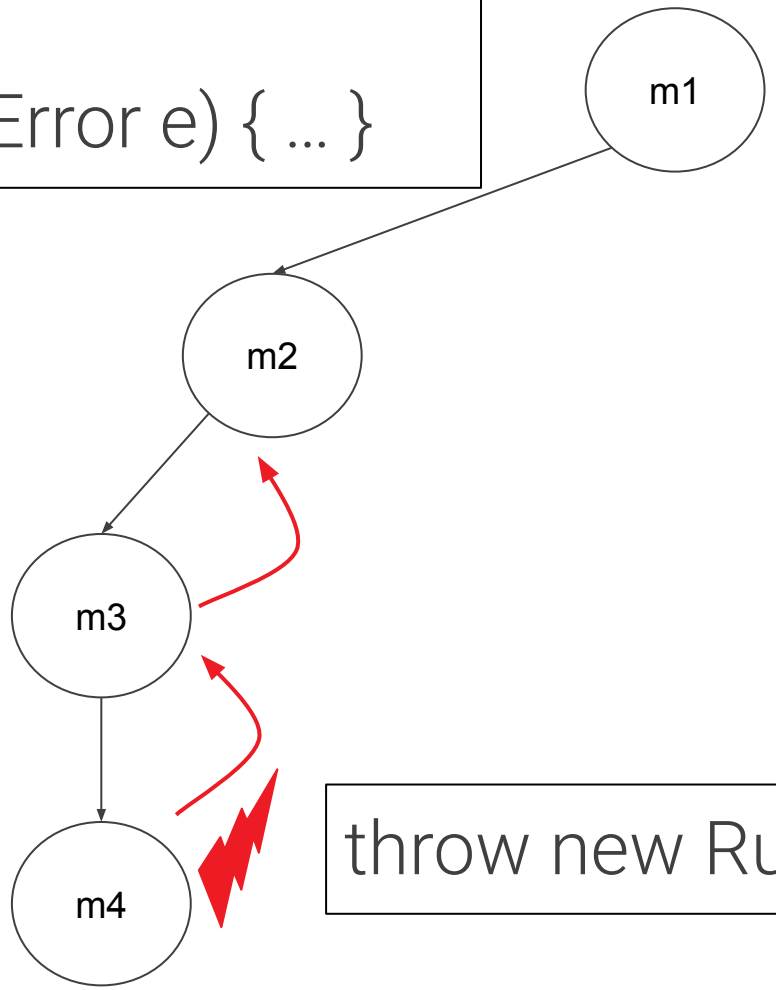
```
try { self m2 }
catch (RuntimeError e) { ... }
```




```
throw new RuntimeError()
```

¿Cómo funcionan?

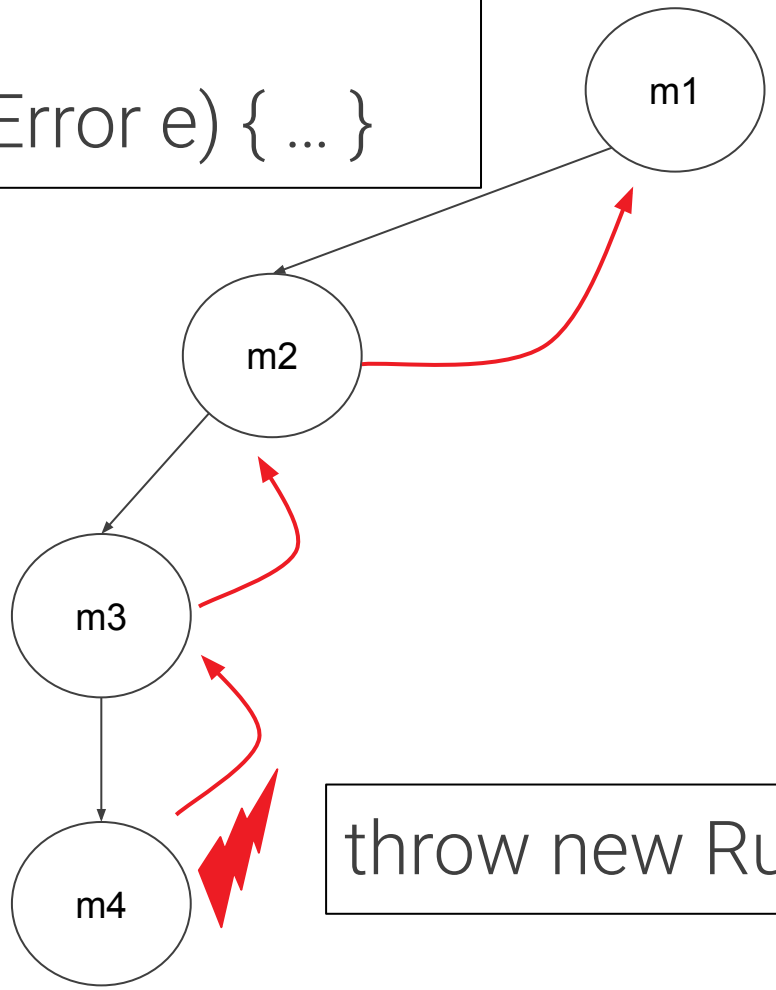
```
try { self m2 }
catch (RuntimeError e) { ... }
```




```
throw new RuntimeError()
```

¿Cómo funcionan?

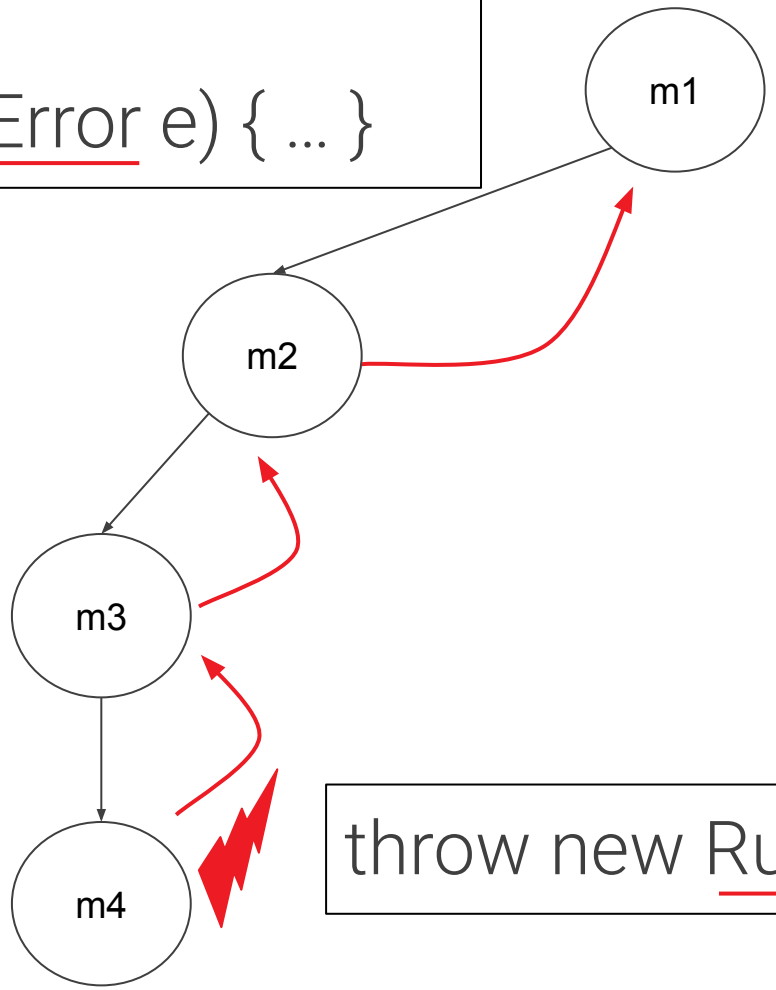
```
try { self m2 }
catch (RuntimeError e) { ... }
```




```
throw new RuntimeError()
```

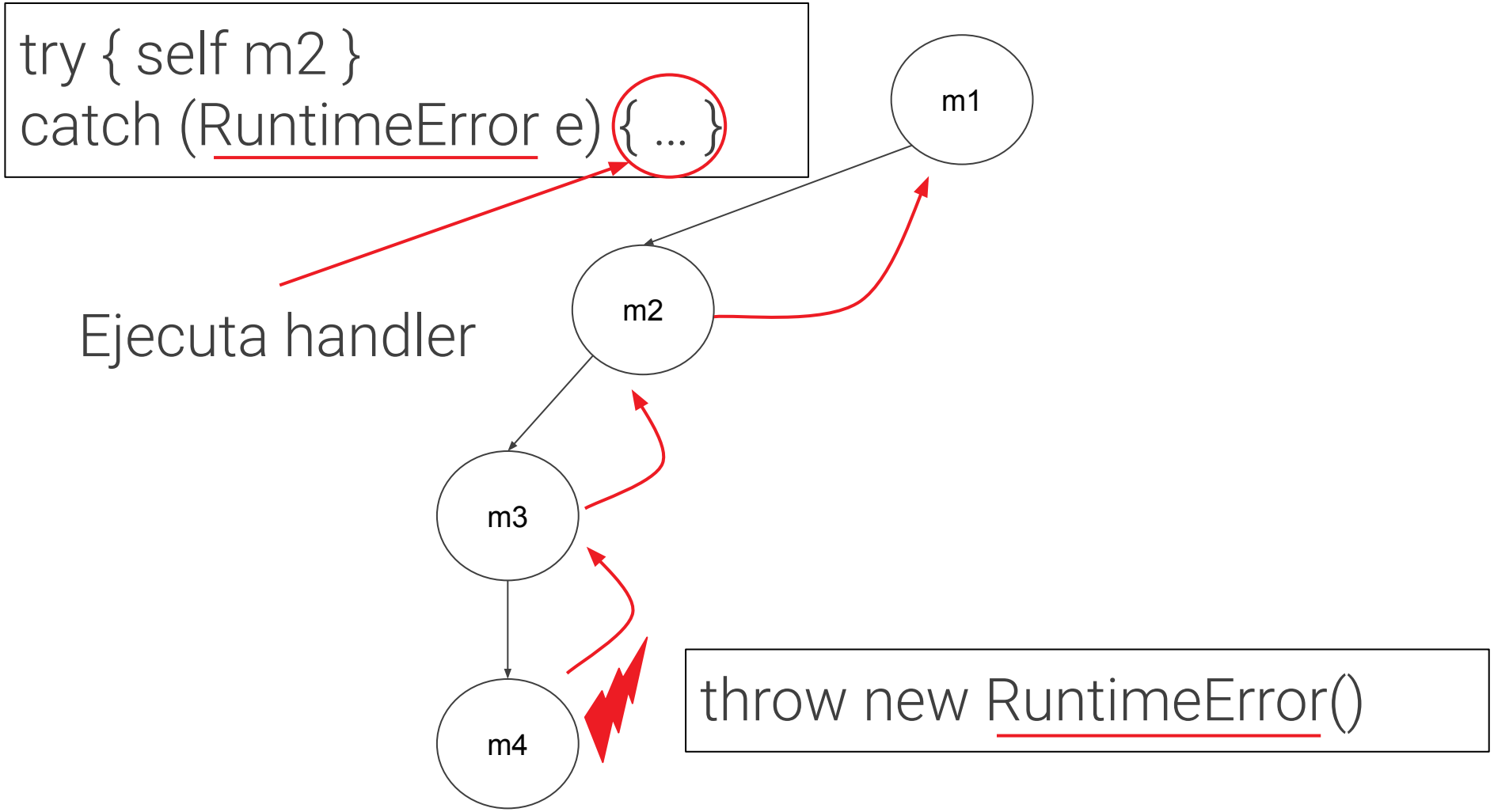
¿Cómo funcionan?

```
try { self m2 }
catch (RuntimeError e) { ... }
```




```
throw new RuntimeError()
```

¿Cómo funcionan?



¿Cómo funcionan?

- Solo ejecuta el handler si se cumple con la condición de handleo
- Se pasa al siguiente handler si no se cumple con la condición de handleo
- Si se llega a la raíz del árbol de ejecución y no hay hander significa que se levantó una “excepción no handleada”

Uso de Excepciones

- ¿**Cuándo levantar** una excepción?
- ¿Quién debe **verificar** que el contrato se cumpla?
- ¿Quién **generalmente las informa**?
- ¿Quién debe **principalmente handlearlas**?
- ¿**Cómo** se puede **handlearlas**?
- ¿**Cómo** se **debe handlearlas**?
- ¿**Qué excepción** informar?
- ¿Usar **checked o unchecked** exceptions?

Cuándo levantar una excepción

- Cuando se rompe un contrato, en particular una pre-condición
- Pensar un pre-condición como algo que no pertenece al dominio de una función matemática
  - Ej: El 0 no está en el dominio de la división
- No levantar excepciones para control de flujo
  - Ej: Para salir de un ciclo, salir de una recursión, etc.

https://web.archive.org/web/20140430044213/http://c2.com/cgi-bin/wiki?DontUseExceptionsForFlowControl

¿Quién debe verificar que el contrato se cumpla?

Escuela C
El objeto que envía *el mensaje/función llamadora debe asegurar las pre-condiciones*

Escuela Lisp
El objeto que recibe *el mensaje/función llamada debe asegurar la pre-condiciones*

¿Quién debe verificar que el contrato se cumpla?

Escuela C

- Ventajas: Performance?
- Desventajas: Código repetido/Inseguridad

Escuela Lisp

- Ventajas: Validaciones en un lugar/Seguridad
- Desventajas: Performance?

¿Quién debe verificar que el contrato se cumpla?

Escuela C

El objeto que envía *el mensaje/función llamadora debe asegurar las pre-condiciones*

Escuela Lisp

El objeto que recibe *el mensaje/función llamada debe asegurar la pre-condiciones*

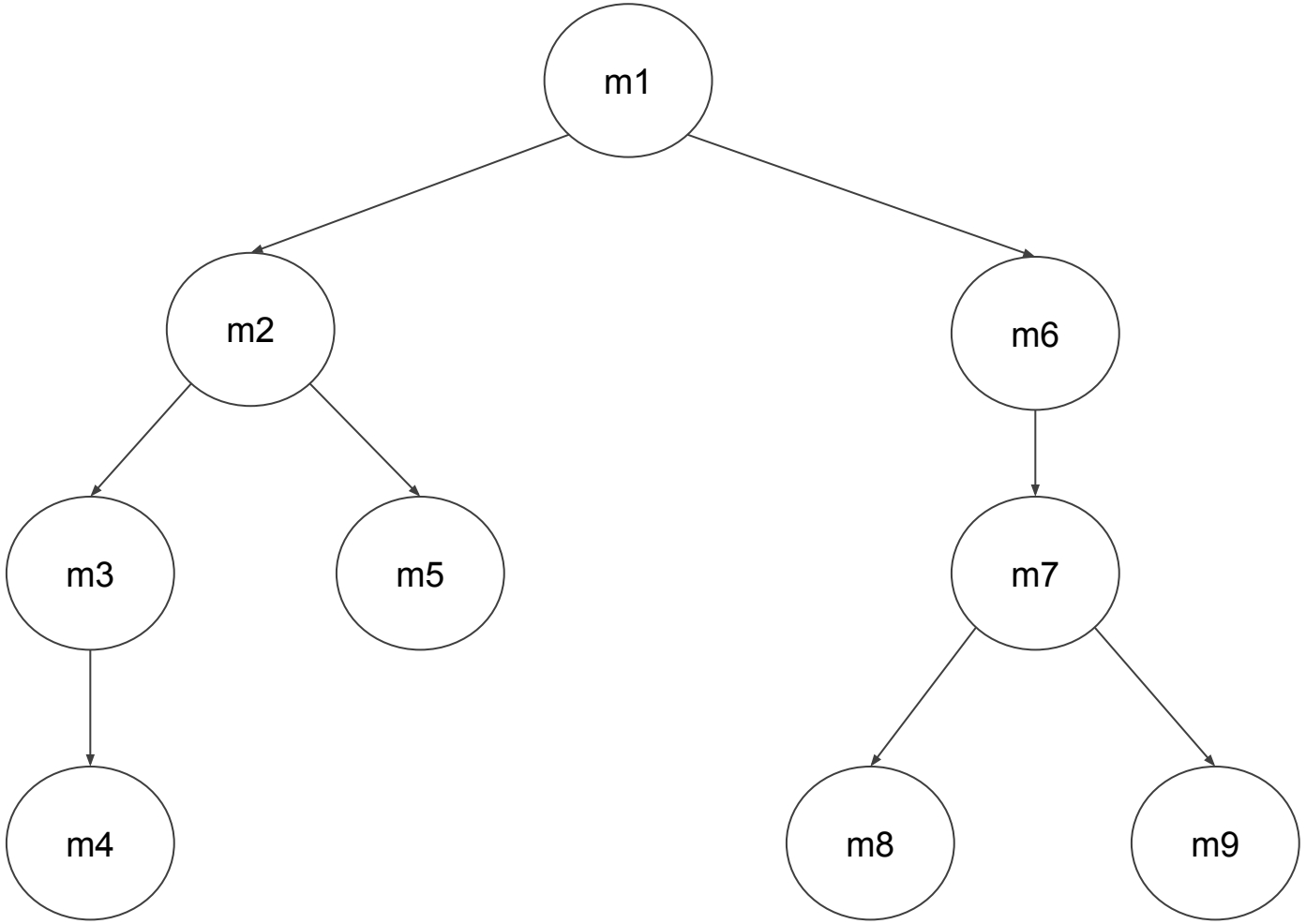
Para pensar…

¿Por qué se usa tanto (más?) la “Escuela C”?

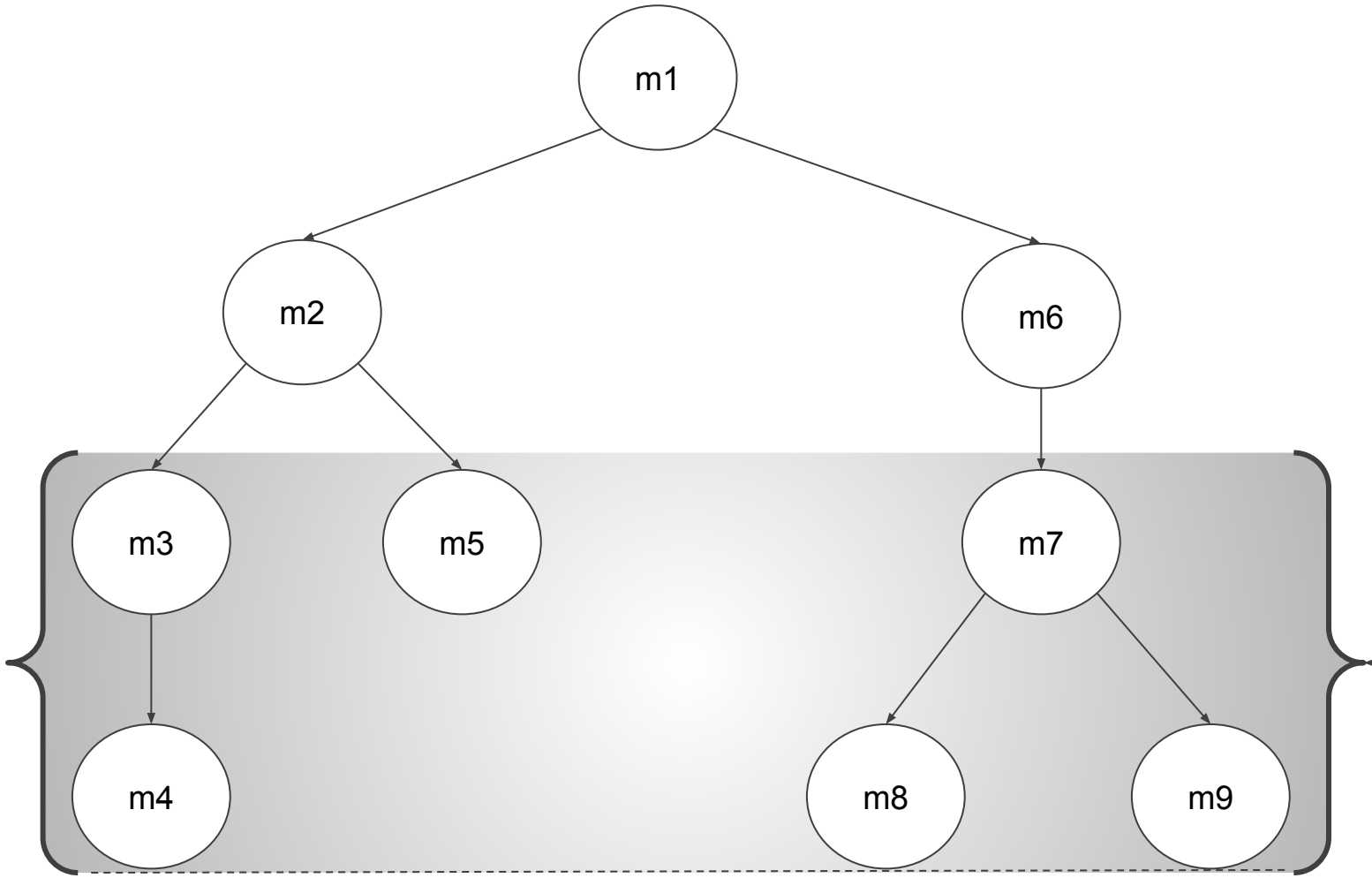
¿Qué condiciones se deben cumpliar para que no haya peligro con la “Escuela C”?

¿Quién generalmente las informa?

Árbol de Ejecución

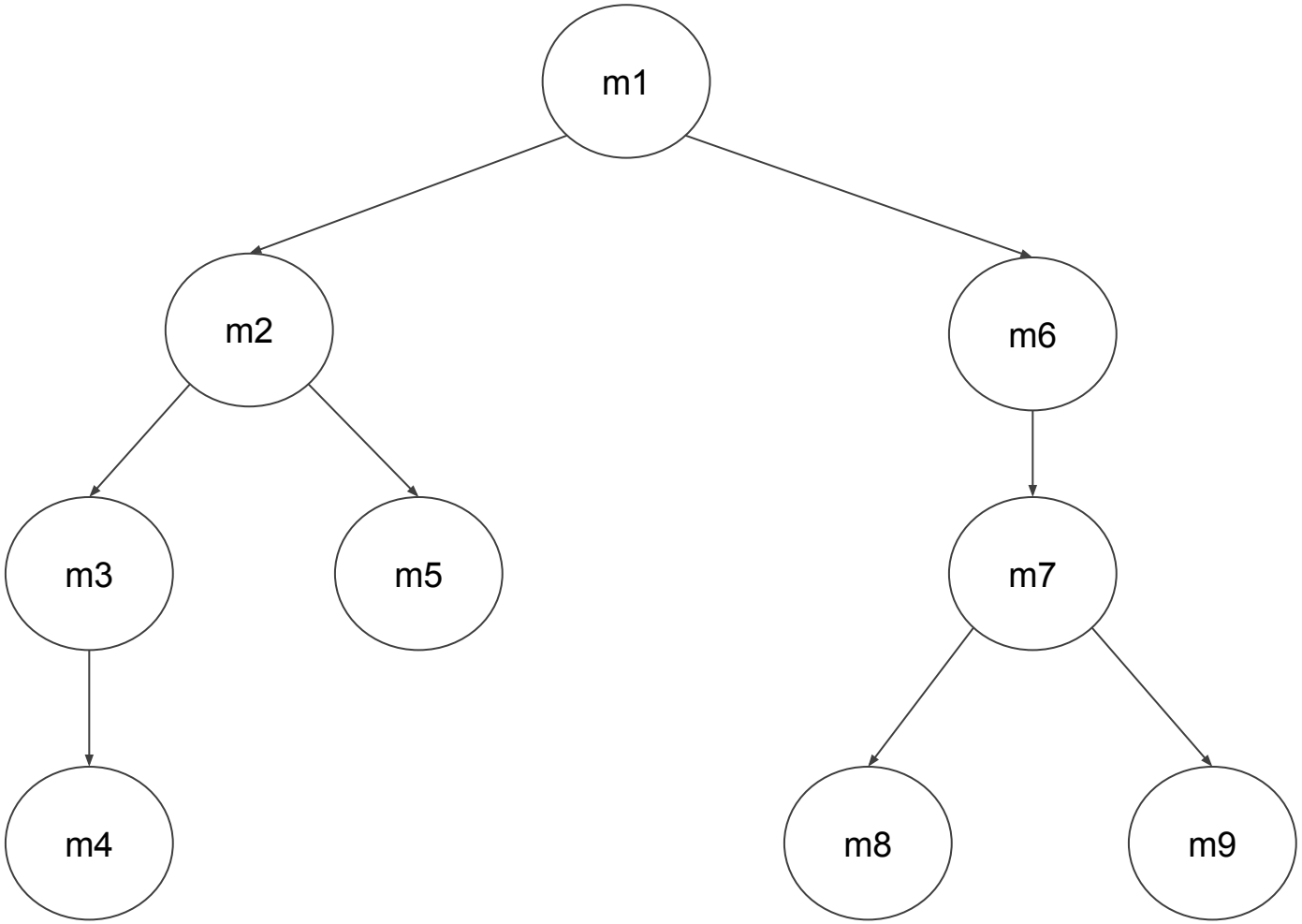


¿Quién generalmente las informa?

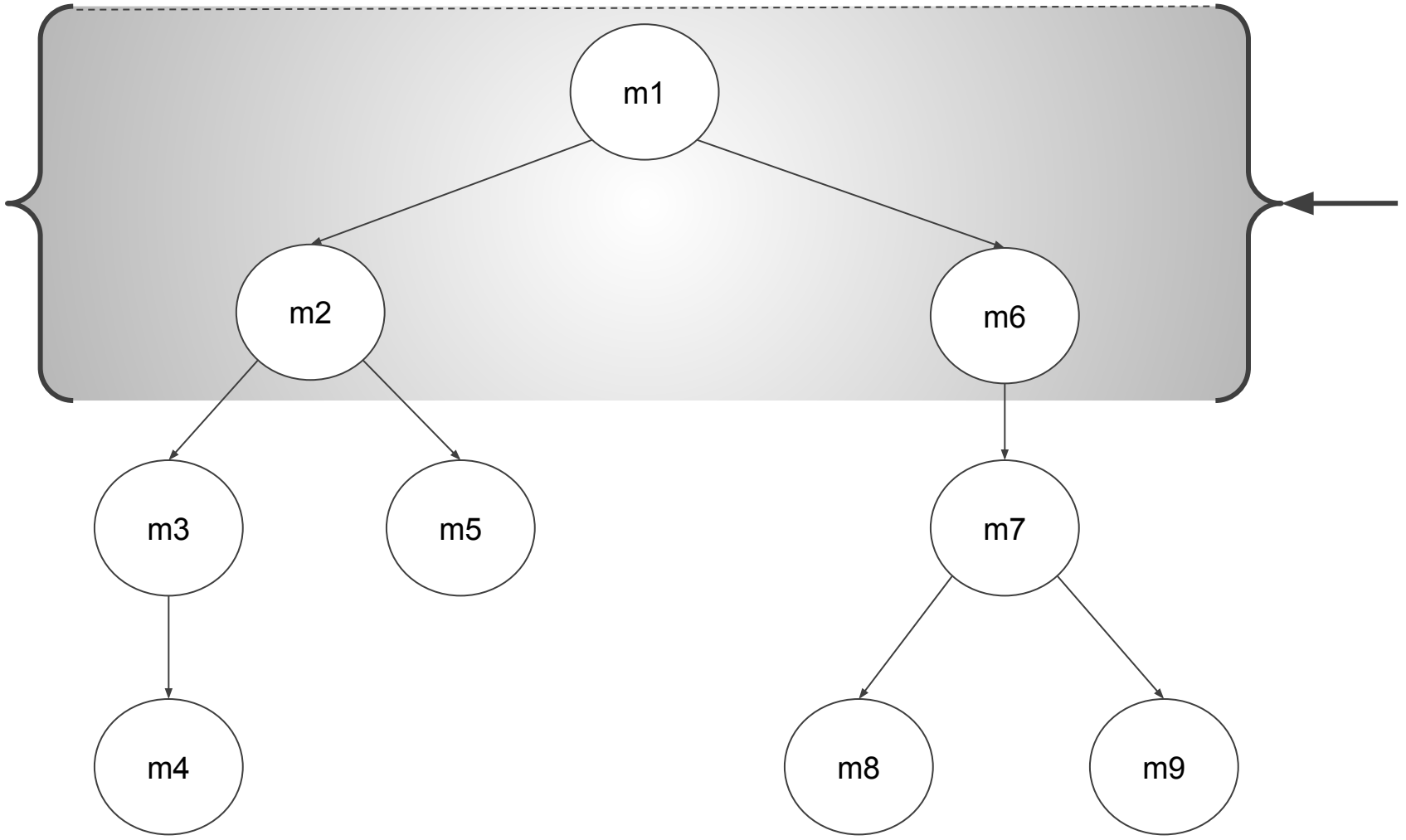


Los objetos que se encuentran más abajo en el árbol de ejecución porque son los que “realmente hacen algo”

¿Quién debe principalmente handlearlas?



¿Quién debe handlearlas?



Los objetos que se encuentran más arriba en el árbol de ejecución ya que tienen más contexto de qué se está haciendo y por lo tanto puede decidir mejor qué hacer

Cómo se **puede** handlearlas

- En implementaciones “cerradas”:
  - Terminar el bloque donde se generó la excepción (bloque del try)
  - Pasar la excepción al siguiente handler
- En implementaciones “abiertas”:
  - Terminar el bloque donde se generó la excepción
  - Pasar la excepción al siguiente handler
  - Reintentar el bloque que generó la excepción
  - Continuar con la siguiente colaboración

Cómo se **debe** handlearlas

- Solo se debe handlear una excepción si se puede resolver la ruptura del contrato
  - Generalmente excepciones de “capas inferiores” indican errores de programación. Prevenirlas para darle semántica en la capa actual (ej. de importación de csv)
- No handlear excepciones si no se puede hacer nada con ellas
  - ¿Qué se puede hacer con un división por 0?
  - ¿Qué se puede hacer con un index out of bound?
- ¡Nunca ocultar las excepciones!
- Handlear todas las excepciones en la raíz del árbol de ejecución

Qué excepción informar

1. Un tipo de excepción por cada condición
   Ej: *IndexOutOfBounds, InvalidBalance, InvalidName*, etc.
2. La misma excepción siempre, no importa la condición de error
   Ej: Usar siempre *RuntimeException* (Java), *Error* (Smalltalk)
3. Un mix

Qué excepción informar

1. Un tipo de excepción por cada condición
   - **Ventaja**: Se por el nombre de la clase qué falló
   - **Desventaja**: Jerarquía enorme de Excepciones
2. La misma excepción siempre, no importa la condición de error
   - **Ventaja**: Jerarquía simple de Excepciones
   - **Desventaja**: El nombre no alcanza para saber qué falló
3. Un mix: ¿Cuándo?

Qué excepción informar

- Lo que determina la **necesidad de existir** de un tipo de excepción es si ***se la handlea o no***
- Esto se debe al acoplamiento que existe en la condición de handleo (catch/on:)
- Solo crear nuevos tipos de excepciones si se los va a handlear
  → Depende del tipo de software under dev.
- Ampliar la condición de handler
  → ej. CuisSmalltalk

Qué excepción informar según tipo de Software

- Desarrollo Propio - Soy dueño del código
  - Crear excepciones sólo cuando es necesario handlearlas
  - No es un problema porque soy dueño del código
- Framework/Librería de Clases
  - Hay que anticipar qué excepciones se quieren handlear
  - Se crean más excepciones “por si acaso”
  - Estamos acostumbrados a esta opción porque es la que vemos

Qué excepción informar - Ampliar Condición

Ejemplo de Cuis Smalltalk

Checked vs. Unchecked Exceptions

- Idea implementada en Java
- Inicialmente parecía una buena idea
  - Hace explícito qué puede fallar
  - Type Safety
  - ¿Error de Programación vs. Error Funcional?
    → ¡Es que es relativo!
- Genera acoplamiento muy fuerte
  - Si se agrega o borra una excepción hay que “recompilar” todo
- Termina ofuscando el código

https://phauer.com/2015/checked-exceptions-are-evil/#no-recovery-possible-at-all

https://wiki.c2.com/?TheProblemWithCheckedExceptions

Implementación

- Disponibilidad de la implementación
  - Abierta vs. Cerrada
- Tratamiento del Stack de Ejecución
  - Deshacer vs. Mantener

Implementación según Disponibilidad de Implementación

- Cerrada, en el runtime del lenguaje:
  - Java, C#, Python, Ruby, etc.
  - No se puede ni ver ni modificar la implementación a menos que sepas de implementación de lenguajes o VMs, etc.
- Abierta, usando el mismo lenguaje:
  - Smalltalk, CommonLisp, Self
  - Se puede ver la implementación
    → se puede aprender de ella
  - Se puede ampliar las prestaciones
    → Ej. CuisSmalltalk

Implementación según Tratamiento de Stack de Ejecución

- Deshacer el Stack:
  - Java, C#, Python, Ruby, etc.
  - Se pierde el contexto de donde se levantó la excepción
  - Fuerza la generación de logs
  - Fuerza que la excepción guarde de manera textual el stack
  - Dificulta el debugging
- Mantener el Stack:
  - Smalltalk, CommonLisp, Self
  - No se pierde el contexto donde fue levantada la excepción
  - → Se puede hacer mejor debugging
  - → Se puede dumpear el stack y revivirlo!

¿Código de Retorno o Excepciones?

Why does Go not have exceptions?

We believe that coupling exceptions to a control structure, as in the `try-catch-finally` idiom, results in convoluted code. It also tends to encourage programmers to label too many ordinary errors, such as failing to open a file, as exceptional.

Go takes a different approach. For plain error handling, Go's multi-value returns make it easy to report an error without overloading the return value. A canonical error type, coupled with Go's other features, makes error handling pleasant but quite different from that in other languages.

Go also has a couple of built-in functions to signal and recover from truly exceptional conditions. The recovery mechanism is executed only as part of a function's state being torn down after an error, which is sufficient to handle catastrophe but requires no extra control structures and, when used well, can result in clean error-handling code.

¿Código de Retorno o Excepciones?

- Explicación poco técnica y sin ningún tipo de datos concreto técnico
- En Go hay que poner ifs por todos lados y ya sabemos lo que significa…

https://eiriktsarpalis.wordpress.com/2017/02/19/youre-better-off-using-exceptions/

¿Either o Excepciones?

- Mismo problema que error code solo que “type safe” :-)
- Pokemon anti-pattern (parecido a checked exceptions)

https://dev.to/anthonyjoeseph/either-vs-exception-handling-3jmg#why-use-either
https://eiriktsarpalis.wordpress.com/2017/02/19/youre-better-off-using-exceptions/

Conclusiones

Conclusiones

- Usar Excepciones para indicar que se “Rompió un Contrato”
- No ocultar excepciones en el handleo
- Crear Excepciones por demanda
- Usar Objetos Válidos para crear seguridad

Bibliografía

- Aprendiendo Smalltalk en época de Cuarentena, episodio de Excepciones: https://youtu.be/7wP-AuinuQk

- Implementando Excepciones en Ruby: https://youtube.com/playlist?list=PLMkq_h36PcLA4yY58tQgj5FAXRzMaZAaY

Bibliografía

- Christophe Dony:
  - A Fully Object-Oriented Exception Handling System: Rationale and Smalltalk Implementation
  - Exception Handling and Object-Oriented Programming: towards a synthesis
  - Improving exception handling with Object-Oriented Programming