Exceptions

Erfan Arvan

Agenda

- Why exceptions?
- Syntax and informal semantics
- Semantic analysis (i.e., type checking rules)
- Operational semantics
- Code Generation for Exception

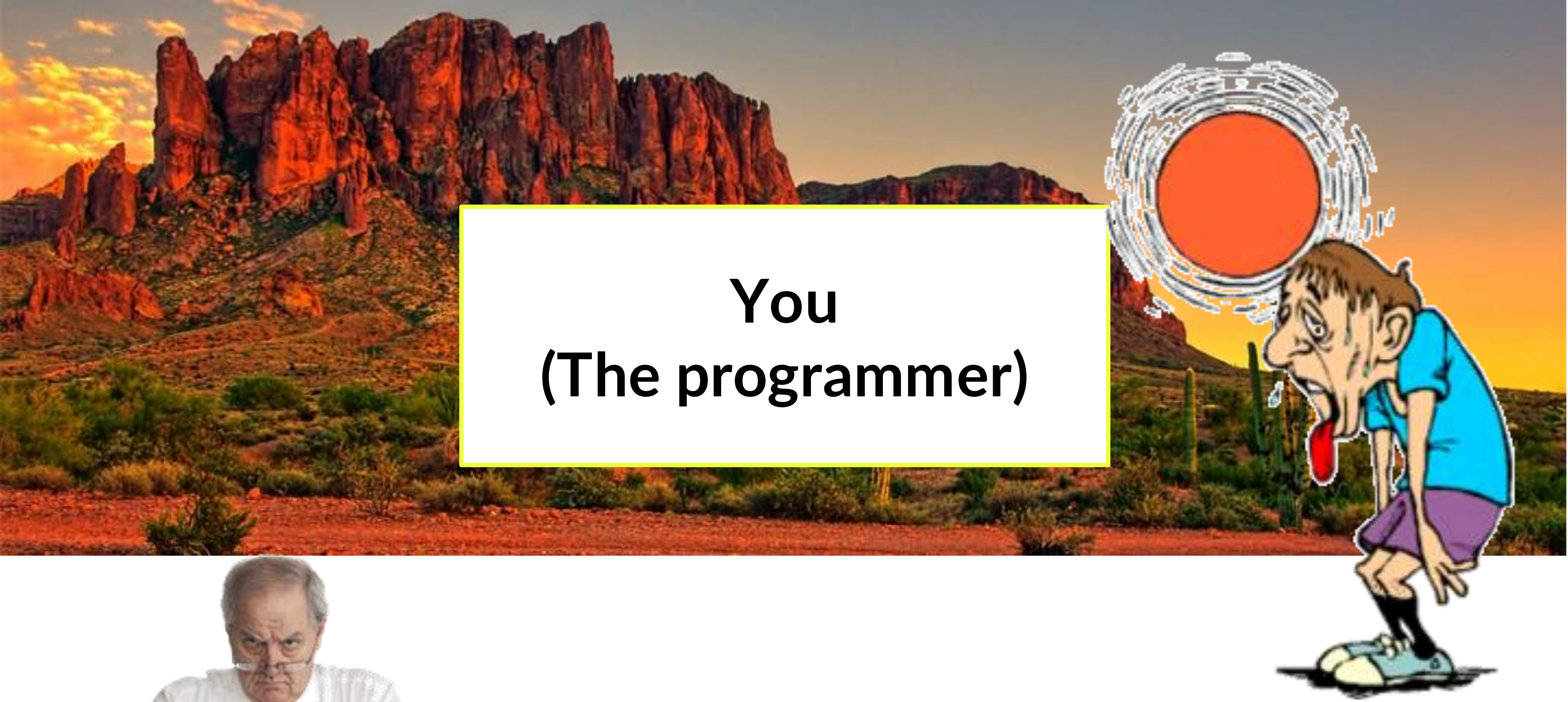
Which is the hottest city in the U.S.?

Which is the hottest city in the U.S.?

Phoenix, Arizona

Which is the hottest city in the U.S.?

Phoenix, Arizona



You
(The programmer)

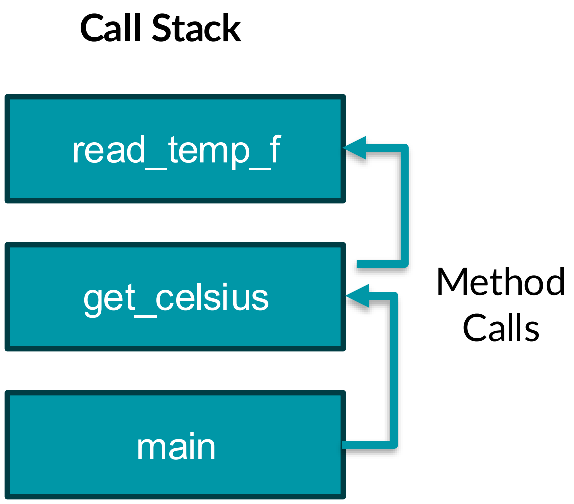
- Implement read_temp_f() to simulate reading the temperature in Phoenix, Arizona (in Fahrenheit).
- Implement get_celsius() to:
    1- First call read_temp_f() to get the temp
    2- Then convert that value to Celsius
- In main(), call get_celsius() and print the result in Celsius.

What is the first thing you should do?

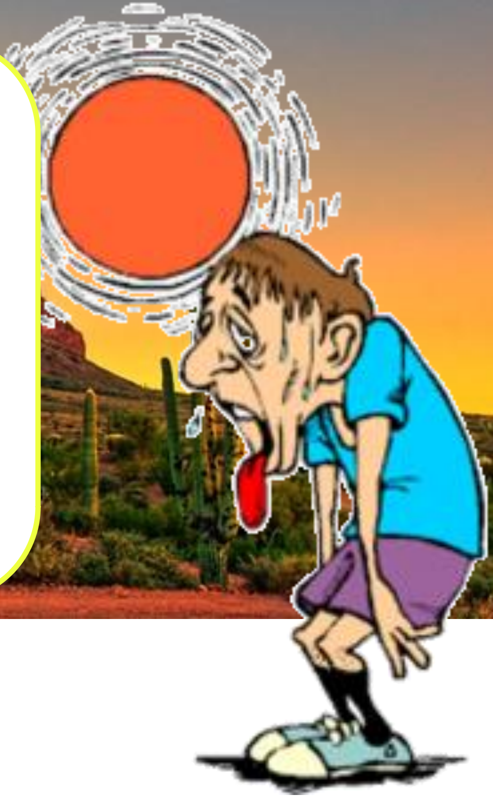
Cool down!

- Implement read_temp_f() to simulate reading the temperature in Phoenix, Arizona (in Fahrenheit).
- Implement get_celsius() to:
  1- First call read_temp_f() to get the temp
  2- Then convert that value to Celsius
- In main(), call get_celsius() and print the result in Celsius.

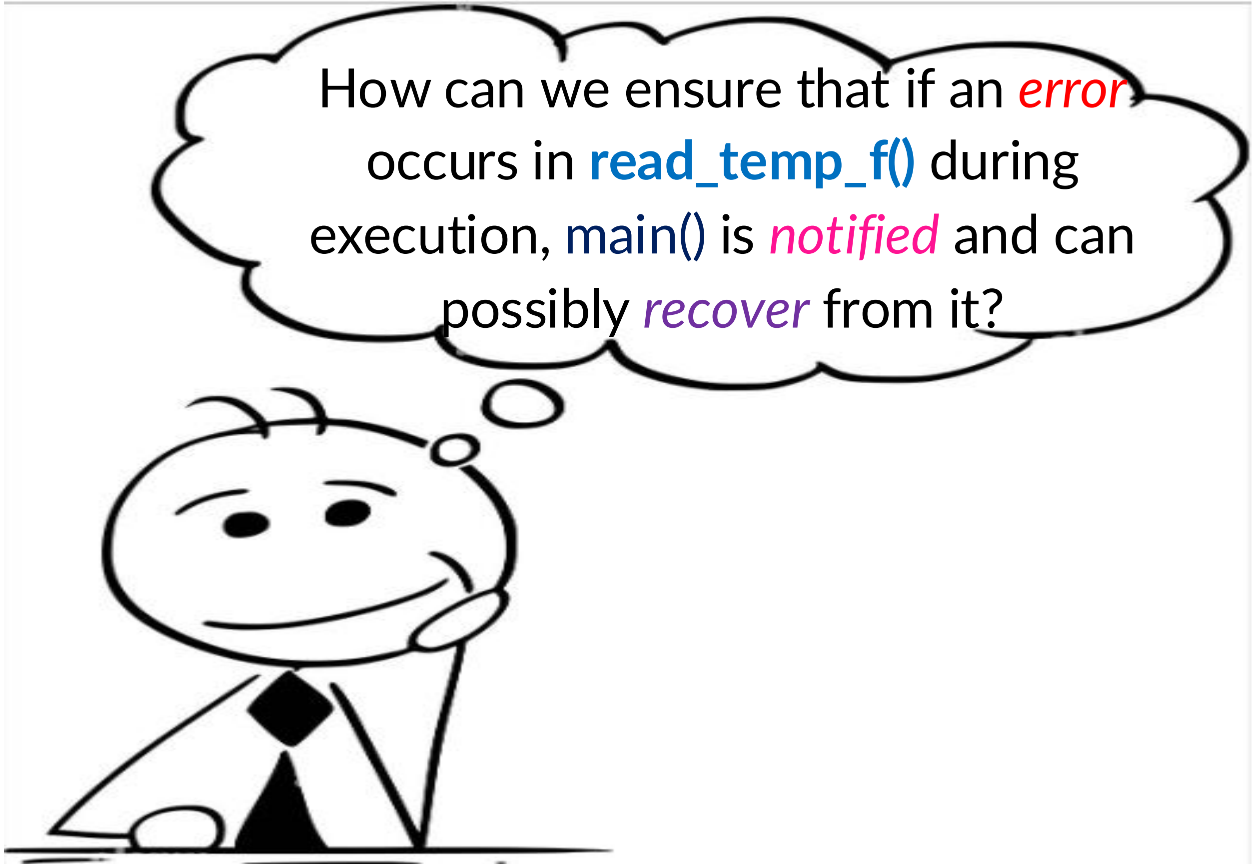
```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
        printf("Temperature: %d°F\n", result);
}
```

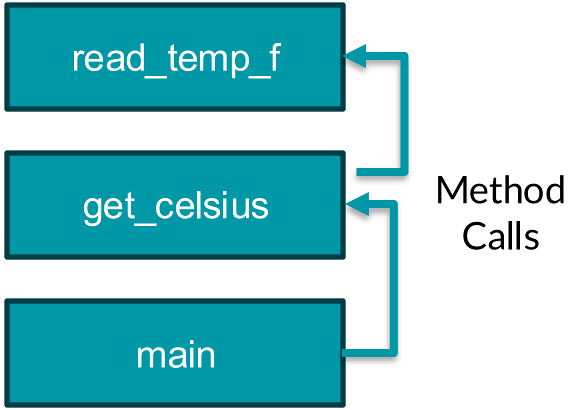


- ❑ The temperature read from sensors in read_temp_f may be faulty and raise Error_In_Sensor.
- ❑ **Make sure to handle sensor failure gracefully in main().**



Main Challenge





Option1: Error Return Codes

- Select special error codes in read_temp_f() and get_celsius()
- When an error happens, the *callee* returns the code to the *caller*
- The *caller* promises to check the error return

and either:
  - Correct the error, or
  - Pass it on to its own caller.
- Very common in C and C++ programming
- Example:
  - malloc() (memory allocation)
    - Returns NULL if memory cannot be allocated.
    - Have you ever checked the return value of malloc() when using it?

```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return ???; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (???) { // read_temp_f failed
        return ???;         // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (???) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

We need to choose error return codes that are distinguishable from normal values

Which is the hottest city in the U.S.?

Phoenix, Arizona

```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return -1.0f; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius < 0.0f) { // read_temp_f failed
        return -20;        // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result==-20) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

```
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return -1.0f; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius < 0.0f) { // read_temp_f failed
        return -20;          // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result==-20) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

Different error conventions

```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return -1.0f; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius < 0.0f) { // read_temp_f failed
        return -20;        // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result==-20) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

Manual error checking in callers

```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return -1.0f; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius < 0.0f) { // read_temp_f failed
        return -20;         // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result==-20) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

Callers need to remember the codes

```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return -1.0f; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius < 0.0f) { // read_temp_f failed
        return -20;         // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result==-20{
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

Manual error checking everywhere

```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return -1.0f; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius < 0.0f) { // read_temp_f failed
        return -20;         // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result==-20) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

Silent errors if a check is forgotten

```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return -1.0f; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius < 0.0f) { // read_temp_f failed
        return -20;         // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result==-20) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

Much Extra Code and Messy!

- ❑ Extend the code to work in Alaska as well.

New Challenge!

```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return -1.0f; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius < 0.0f) { // read_temp_f failed
        return -20;        // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result==-20) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

```c
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return -1.0f; // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius < 0.0f) { // read_temp_f failed
        return -20;         // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result==-20) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

Error codes can now be *legitimate temperatures*!

```
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return std::numeric_limits<float>::min(); // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius == std::numeric_limits<float>::min()) { // read_temp_f failed
        return std::numeric_limits<int>::min(); // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result == std::numeric_limits<int>::min(); ) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

Error codes can now be *legitimate temperatures*!

```
// read_temp_f: reads temperature from a sensor
float read_temp_f() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        return std::numeric_limits<float>::min(); // Error code
    }
    return temperature;
}

// get_celsius: converts Celsius to Fahrenheit, needs valid temperature
int get_celsius() {
    float celsius = read_temp_f();
    if (celsius == std::numeric_limits<float>::min()) { // read_temp_f failed
        return std::numeric_limits<int>::min();  // get_celsius own error code
    }
    int fahrenheit = (int)(celsius * 9.0f / 5.0f + 32.0f);
    return fahrenheit;
}

// main: gets Fahrenheit temperature or prints error
int main() {
    int result = get_celsius();
    if (result == std::numeric_limits<int>::min(); ) {
        printf("Error reading temperature!\n");
    } else {
        printf("Temperature: %d°F\n", result);
    }
}
```

Error codes can now be *legitimate temperatures*!

Hard to change or extend

Option1: Error Return Codes

- **Problems?**
  - It might be hard to select a value as an error code

    e.g., double sum (double num1, double num2)

    How can we handle this?
  - Different error codes, hard to remember, and extend
  - It is easy to forget to check the error return codes
  - Much extra code, messy…

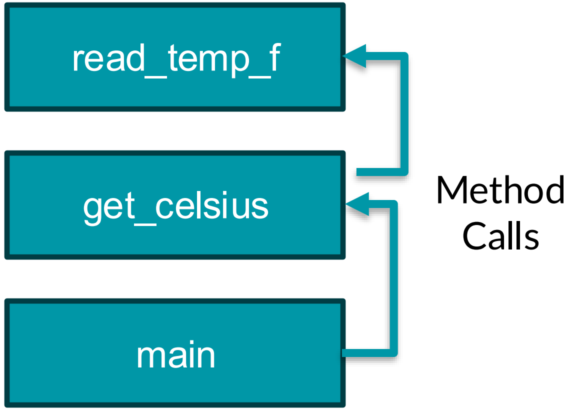
Option2: Exceptions

Option2: Exceptions

- Exceptions are a language mechanism designed to allow:
    - Deferral of error handling to a *caller*
    - Without (explicit) *error codes* 
    - And without (explicit) error return code checking

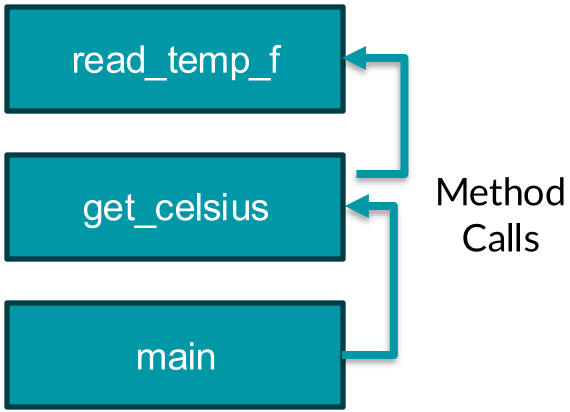
```cpp
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Less Extra Code



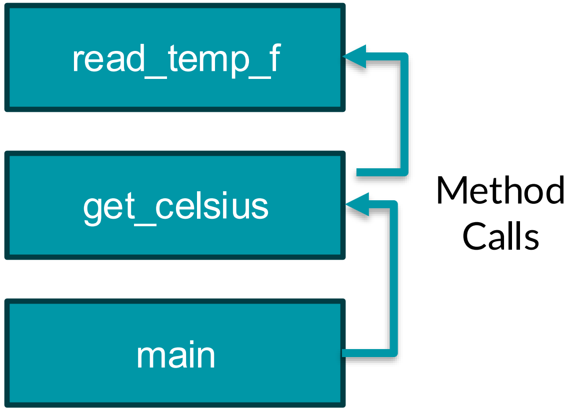
```cpp
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Less Extra Code



```cpp
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

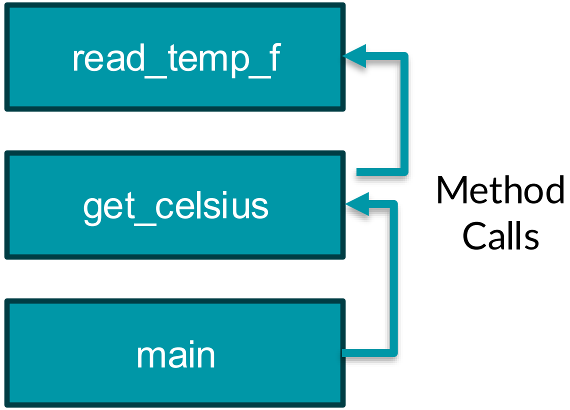
Less Extra Code



```cpp
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Normal **execution STOPS**

Less Extra Code

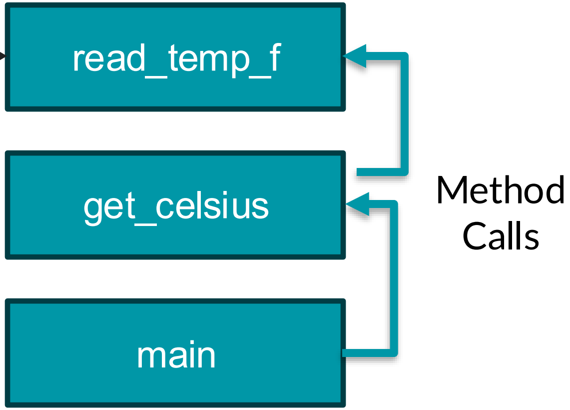


```cpp
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Normal **execution STOPS**

Less Extra Code

The runtime system looks for a catch

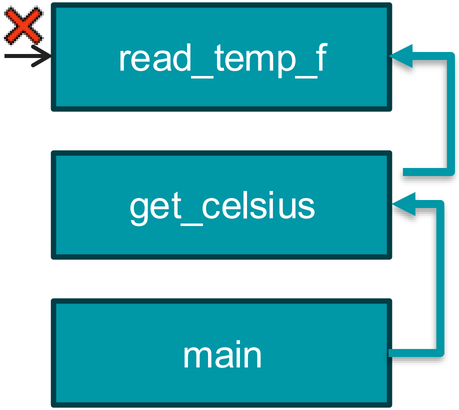


```cpp
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Normal **execution STOPS**

Less Extra Code

The runtime system looks for a catch



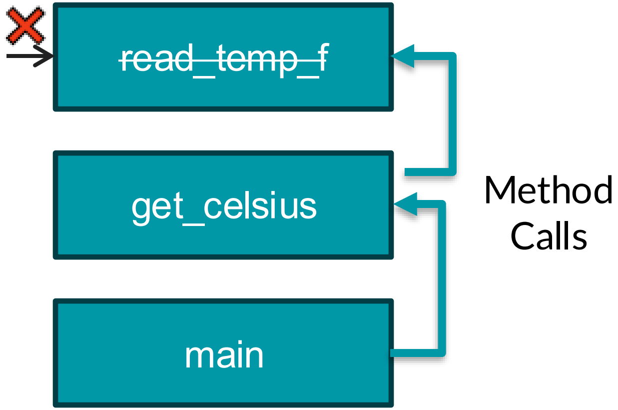
Method Calls

```
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Less Extra Code

Normal **execution STOPS**

The runtime system looks for a catch



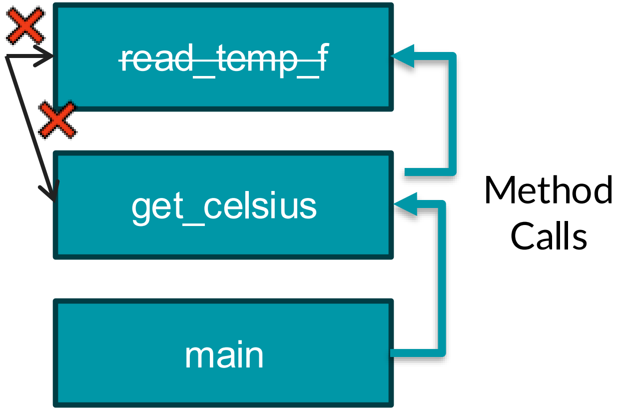
Unwind: Read_temp_f() does not catch the exception, so Read_temp_f() 's execution is abandoned — variables in Read_temp_f() are destroyed, memory is cleaned up, and it "pops off" the call stack.

```
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Normal **execution STOPS**

Less Extra Code

The runtime system looks for a catch



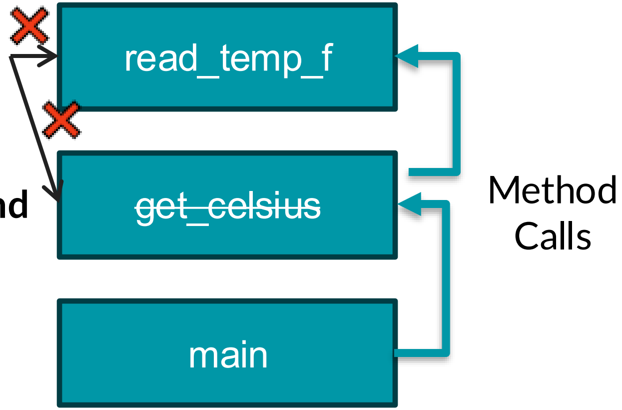
```cpp
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Normal **execution STOPS**

Less Extra Code

The runtime system looks for a catch

Unwind

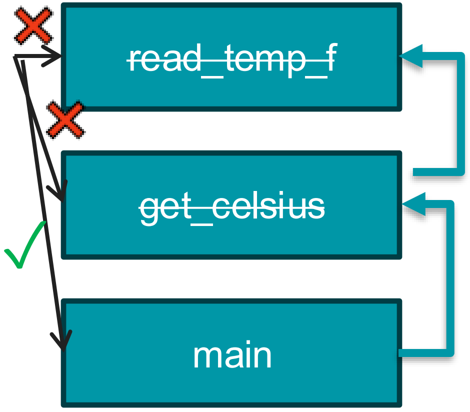


```cpp
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Normal **execution STOPS**

Less Extra Code

The runtime system looks for a catch

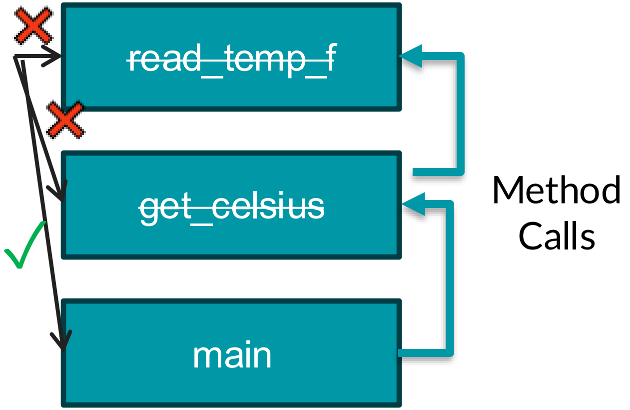


Method Calls

```cpp
#include <cstdio>
#include <stdexcept>

float bar() {
    float temperature = /* get temp from sensor*/;
    if (Error_In_Sensor) {
        throw std::runtime_error("Error in bar()");
    }
    return 42.0f;
}

int foo() {
    float val = bar();
    return (int)(val + 1);
}

int main() {
    try {
        int result = foo();
        printf("Result: %d\n", result);
    } catch (const std::runtime_error& e) {
        printf("Caught error: %s\n", e.what());
    }
}
```

Less Extra Code

The runtime system looks for a catch



Control jumps here

Adding Exceptions to Cool

- We extend the language of expressions:

  $$e :: throw\ e \mid try\ e\ catch\ x{:}T \ \Longrightarrow e_2$$

- (Informal) semantics of $throw\ e$
  - Signals an exception
  - *Interrupts* the current evaluation and *searches* for an exception handler up the activation tree
  - The value of e is an exception parameter and can be used to communicate details about the exception

Typing Exceptions (1)

- We must extend the Cool typing judgment

$$O, M, C \vdash e : T$$

  - Type T refers to the normal return value!
- We'll start with the rule for try:
  - Parameter "x" is bound in the catch expression – try is like a conditional

$$\frac{O, M, C \vdash e : T_1 \qquad O[T/x], M, C \vdash e' : T_2}{O, M, C \vdash \text{try } e \text{ catch } x : T \Rightarrow e' : T_1 \sqcup T_2}$$

Typing Exceptions (2)

- What is the type of “throw e” ?
- The type of an expression:
  - Is a description of the possible return values, and
  - Is used to decide in what contexts we can use the expression

“throw” does not return to its immediate context but directly to the exception handler!

- The same “throw e” is valid in any context:

  if throw e then (throw e) + 1 else (throw e).foo()

As if “throw e” has *any type*!

Typing Exceptions(3)

$$\frac{O, M, C \vdash e : T_1}{O, M, C \vdash \text{throw } e : T_2}$$

- As long as “e” is well typed, “throw e” is well typed with any type needed in the context.
  - T_2 is unbound!

- This is convenient because we want to be able to signal errors from any context

Tired?

Operational Semantics of Exceptions

- Several ways to model the behavior of exceptions
- A generalized value is
  - Either a normal termination value, or
  - An exception with a parameter value

    g ::= Norm(v) | Exc(v)
- Thus given a generalized value we can:
  - Tell if it is normal or exceptional return, and
  - Extract the return value or the exception parameter

Operational Semantics of Exceptions (1)

- The existing rules change to use Norm(v) :

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : Norm(Int(n_1)), S_1 \\ so, E, S_1 \vdash e_2 : Norm(Int(n_2)), S_2 \end{array}}{so, E, S \vdash e_1 + e_2 : Norm(Int(n_1 + n_2)), S_2}$$

$$\frac{\begin{array}{c} E(id) = l_{id} \\ S(l_{id}) = v \end{array}}{so, E, S \vdash id : Norm(v), S}$$

$$\frac{}{so, E, S \vdash self : Norm(so), S}$$

Operational Semantics of Exceptions (2)

- "throw" returns exceptionally:

$$\frac{so, E, S \vdash e : v, S_1}{so, E, S \vdash \text{throw } e : \text{Exc}(v), S_1}$$

- The rule above *is not well formed*! Why?

Operational Semantics of Exceptions (2)

- "throw" returns exceptionally:

$$\frac{so, E, S \vdash e : v, S_1}{so, E, S \vdash \text{throw } e : Exc(v), S_1}$$

- The rule above *is not well formed*! Why? We want:

$$\frac{so, E, S \vdash e : Norm(v), S_1}{so, E, S \vdash \text{throw } e : Exc(v), S_1}$$

Operational Semantics of Exceptions (3)

- "throw e" always returns exceptionally:

$$\frac{so, E\,,\,S \vdash e : Norm(v), S_1}{so, E, S \vdash \text{throw } e : Exc(v), S_1}$$

- What if the evaluation of e itself throws an exception?
  - Informally: "throw (1 + (throw 2))" is like "throw 2"
  - Formally:

$$\frac{so, E\,,\,S \vdash e : Exc(v), S_1}{so, E, S \vdash \text{throw } e : Exc(v), S_1}$$

Operational Semantics of Exceptions (4)

- All existing rules are changed to propagate the exception:

$$\frac{so, E, S \vdash e_1 : Exc(v), S_1}{so, E, S \vdash e_1 + e_2 : Exc(v), S_1}$$

  - Note: the evaluation of e_2 is skipped
- What if the evaluation of e itself throws an exception?

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : Norm(Int(n_1)), S_1 \\ so, E, S_1 \vdash e_2 : Exc(v), S_2 \end{array}}{so, E, S \vdash e_1 + e_2 : Exc(v), S_2}$$

Operational Semantics of Exceptions (5)

- The rules for "try" expressions:
  - – Multiple rules (just like for a conditional)

$$\frac{so, E, S \vdash e : Norm(v), S_1}{so, E, S \vdash try\ e\ catch\ x : T \Rightarrow e' : Norm(v), S_1}$$

- What if e terminates exceptionally?
  - We must check whether it terminates with an exception parameter of type T or not.

Operational Semantics of Exceptions (6)

- If e does not throw the expected exception

$$\frac{\begin{array}{c}\text{so}, E, S \vdash e : ??????????\\ v = X(\ldots)\\ \text{Not }(??????????)\end{array}}{\text{so}, E, S \vdash \text{try } e \text{ catch } x : T \Rightarrow e' : \text{Exc}(v), S_1}$$

- If e does throw the expected exception

$$\frac{\begin{array}{c}\text{so}, E, S \vdash e : \text{Exc}(v), S_1\\ v = X(\ldots)\\ X \leq T\\ L_{new} = ??????????\\ \text{So}, ?????????? \vdash e' : g, S_2\end{array}}{\text{so}, E, S \vdash \text{try } e \text{ catch } x : T \Rightarrow e' : g, S_2}$$

Operational Semantics of Exceptions (7)

- If e does not throw the expected exception

$$\frac{\begin{array}{c}\text{so, E, S} \vdash \text{e : Exc(v), S}_1 \\ \text{v = X(...)} \\ \text{not (X} \leq \text{T)}\end{array}}{\text{so, E, S} \vdash \text{try e catch x : T} \Rightarrow \text{e}' \text{ : Exc(v), S}_1}$$

- If e does throw the expected exception

$$\frac{\begin{array}{c}\text{so, E, S} \vdash \text{e : Exc(v), S}_1 \\ \text{v = X(...)} \\ \text{X} \leq \text{T} \\ \text{l}_{\text{new}} = \text{newloc(S}_1\text{)} \\ \text{so, E[l}_{\text{new}}\text{/x] , S}_1\text{[v/l}_{\text{new}}\text{]} \vdash \text{e}' \text{ : g, S}_2\end{array}}{\text{so, E, S} \vdash \text{try e catch x : T} \Rightarrow \text{e}' \text{ : g, S}_2}$$

Operational Semantics of Exceptions: Notes

- Our semantics is precise
- But is not very clean
    - It has two or more versions of each original rule
- It is not a good recipe for implementation
    - It models exceptions as “compiler-inserted propagation of error return codes”
    - There are much better ways of implementing exceptions

Code Generation for Exceptions

- One method is suggested by the operational semantics
- Simple to implement
- But not very good
  - We pay a cost at each call/return (i.e., often)
  - Even though exceptions are rare (i.e., exceptional)
- A good engineering principle:
  - Don’t pay often for something that you use rarely!
    - What is Amdahl’s Law?
- – Optimize the common case!

Code Generation for Exceptions: C

- **No built-in exception handling**
- **Manual / Low-Level with setjmp/longjmp**
- Achieved using:

```
#include <setjmp.h>
jmp_buf env;
if (setjmp(env) == 0) {
    // do some risky job in which longjmp(env, 1); // Throw
} else {
    // Catch block
}
...
```

Code Generation for Exceptions: C

- How it works
  - setjmp() saves stack/context
  - longjmp() jumps back to saved point
- Disadvantages
  - No automatic cleanup of stack variables
  - Programmer must manually manage control flow
  - Prone to bugs, no type safety
  - Unsafe
  - No error types or hierarchy

Exceptions in Java

- *"An exception is an **event** that occurs during the **execution** of a program that **disrupts** the **normal flow** of instructions." (oracle)*
- Exception = Exceptional Event

```
public class X {
  public static void main(String[] args) {
    String str = null;
    printLen(str);
  }
  public static int printLen(String str){
    System.out.println(str.length());
  }
}
```

Exceptions in Java

- *"An exception is an event that occurs during the **execution** of a program that **disrupts** the **normal flow** of instructions." (oracle)*
- Exception = Exceptional Event

```
public class X {
  public static void main(String[] args) {
    String str = null;
    printLen(str);
  }
  public static int printLen(String str){
    System.out.println(str.length());
  }
}
```

This code compiles without any errors

Exceptions in Java

- *"An exception is an **event** that occurs during the **execution** of a program that **disrupts** the **normal flow** of instructions." (oracle)*
- Exception = Exceptional Event

```
1.  public class X {
2.    public static void main(String[] args) {
3.      String str = null;
4.      printLen(str);
5.    }
6.    public static int printLen(String str){
7.      System.out.println(str.length());
8.    }
9.  }
```

This code compiles without any errors

```
Exception in thread "main" java.lang.NullPointerException
at NullPointerExample.main(NullPointerExample.java:4)
```

In runtime

Exceptions in Java

- *"An exception is an event that occurs during the* ***execution*** *of a program that* ***disrupts*** *the* ***normal flow*** *of instructions." (oracle)*
- **Are all Exceptional Events bad?**

Exceptions in Java

- *“An exception is an event that occurs during the* ***execution*** *of a program that* ***disrupts*** *the* ***normal flow*** *of instructions.” (oracle)*
- **Are all Exceptional Events bad?**
  - No! Example?

Exceptions in Java

- *"An exception is an event that occurs during the **execution** of a program that **disrupts** the **normal flow** of instructions." (oracle)*
- **Are all Exceptional Events bad?**
    - No! Example?

```
try {
    FileReader reader = new FileReader("config.txt");
} catch (FileNotFoundException e) {
    // File doesn't exist? Create a new one with defaults
    createDefaultConfigFile();
}
```

Exceptions in Java

- *“An exception is an event that occurs during the* ***execution*** *of a program that* ***disrupts*** *the* ***normal flow*** *of instructions.” (oracle)*
- **Are all Exceptional Events bad?**
  - No! Example?

```
@Test
void testExceptionThrown() {
    assertThrows(IllegalArgumentException.class, () -> {
        someMethodThatShouldThrow();
    });
}
```

Exceptions

- Do you know of any other ***runtime semantic constraint violations*** in Java?

  - Division by zero → ArithmeticException
  - Array index out of bounds → ArrayIndexOutOfBoundsException
  - Stack overflow due to infinite recursion → StackOverflowError
  - Out of memory → OutOfMemoryError

  …

Exceptions in Java

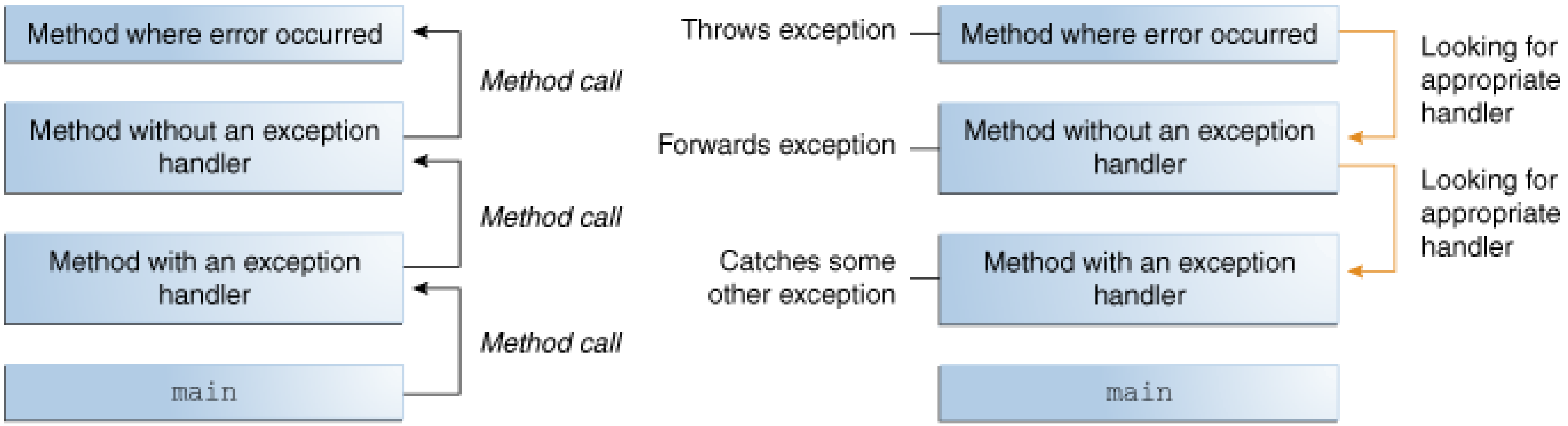
- **Exceptional Events**
  - **Language-Defined Exceptions**
    - Resource Exhaustion (disk full, out of memory, etc.)
    - Invalid Input (e.g., bad method parameters)
    - Runtime Errors (e.g., null pointer dereference)
  - **Custom Exceptions**
    - Must be explicitly thrown
    - Example in Java:

```
throw new InvalidAgeException("Age must be 18 or older.");
```

Exceptions in Java

- **Checked Exceptions**
  - Verified by compiler at compile time.
  - Must be caught or declared with throws.
  - Caused by external issues (files, database, network).
  - Examples: IOException, SQLException, FileNotFoundException
- **Unchecked Exceptions**
  - Not checked by compiler.
  - Caused by programming errors (bugs).
  - *No requirement to catch or declare.*
  - Examples: NullPointerException, ArrayIndexOutOfBoundsException, IllegalArgumentException

Exception Handling in Java



Zero-Cost (Table-Based) Exception Handling

- The Java compiler (javac) generates exception tables in the class bytecode
- Tables map code regions to catch blocks for specific exception types
- During normal execution, the JVM ignores the tables → no overhead
- When an exception is thrown, the JVM:
  - Uses the table to find the correct catch block
  - Unwinds the stack
  - Executes any associated finally blocks
- Ensures structured, type-safe, and efficient error handling
- Called “zero-cost” because there’s no performance impact unless an exception occurs

Exception Handling in Java

- How it works (at runtime):
  - JVM uses precomputed exception tables
  - If throw happens:
    - JVM searches table for *matching handler*
    - Skips over intermediate frames
    - Unwinds stack
    - Transfers control to catch

Code Generation for Exceptions: Java

```java
// try-catch
try { } catch (Exception e) { }

// try-catch-finally
try { } catch (Exception e) { } finally { }

// try-finally (no catch)
try { } finally { }

// throw
throw new IllegalArgumentException("Bad input");

// throws (in method signature)
public void myMethod() throws IOException { }
```