Exceptional Programming

Programming Languages

William Killian

Millersville University

Outline

- Exceptions
- Handling Exceptions
- Constructs
- Case Studies
  - C++
  - Java
  - Python
  - OCaml

Exceptions

- Any unusual, unexpected event that can be *detected* either by hardware or software that usually requires special processing
- This special processing is called *handling the exception*
- The special processing code is called an *exception handler*
- Most languages provide an abstraction around exceptions and exception handlers

Exceptions → Special Events

- In general, event handling logic isn't so different from exception handler logic
- Exceptions are ultimately a special event, of which some amount of information can be included and likewise extracted

What types of "events" would you consider exceptional?

What type of information should you be able to get from an exception?

Exception Terminology

- **Raising** – an exception is raised when its associated event occurs
- **Handling** – processing the exception
- **Continuing** – resuming program execution after handling an exception
- **Fatal exception** – a special class of exception which is unrecoverable
- **Aborting/Terminating** – ending program execution after encountering a fatal exception

Exceptional Control Flow

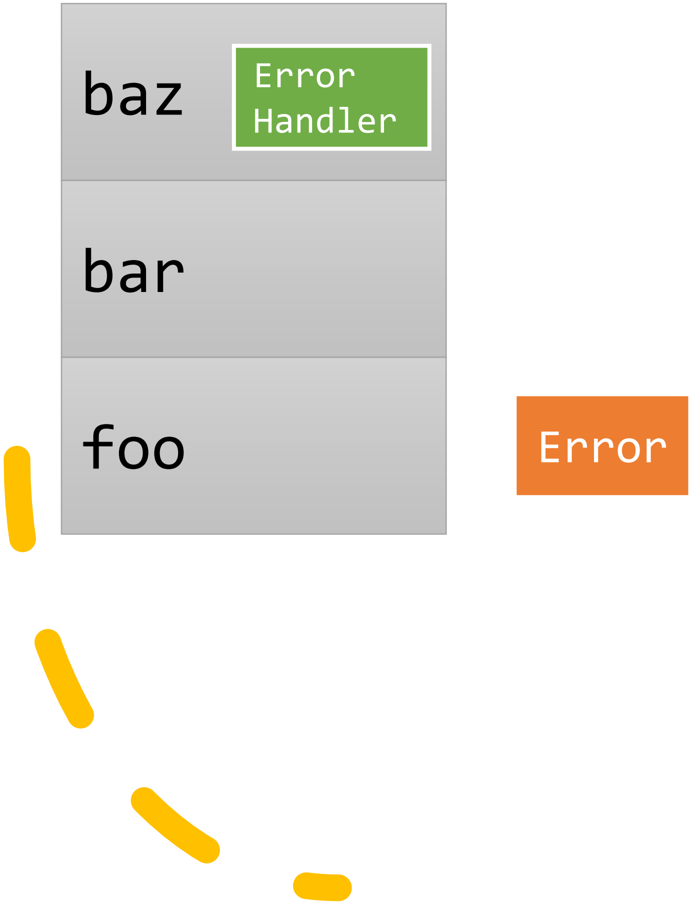
- When an exception occurs, we will walk “up” the activation record stack until we encounter a handler for that exception.

- If we do not encounter an exception handler, then the Operating System must handle it.

```
void foo() {
  raise Error()
}
void bar() {
  foo()
}
void baz() {
  try {
    bar()
  } catch (Error) {

  }
}
```

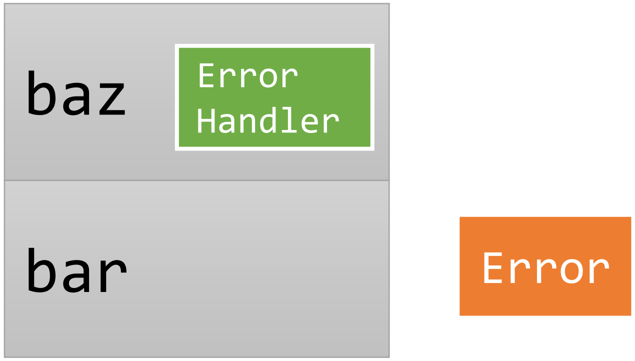
Exceptional Control Flow



```
void foo() {
  raise Error()
}
void bar() {
  foo()
}
void baz() {
  try {
    bar()
  } catch (Error) {

  }
}
```

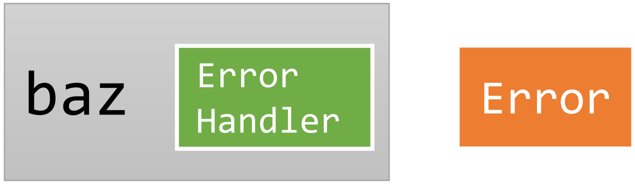
Exceptional Control Flow



```
void foo() {
  raise Error()
}
void bar() {
  foo()
}
void baz() {
  try {
    bar()
  } catch (Error) {

  }
}
```

Exceptional Control Flow

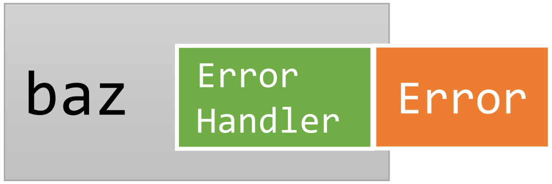


We have <u>unwound</u> the stack to the handler

```
void foo() {
  raise Error()
}
void bar() {
  foo()
}
void baz() {
  try {
    bar()
  } catch (Error) {

  }
}
```

Exceptional Control Flow



We can now invoke the error handling code

```
void foo() {
  raise Error()
}
void bar() {
  foo()
}
void baz() {
  try {
    bar()
  } catch (Error) {

  }
}
```

Alternatives to Exceptions?

- Use the return value to indicate error
- Use an out-parameter to indicate error
- Pass an error-handling subprogram as a parameter

What else could / would you use?

Handling Exceptions

Handling Exceptions

- What information should we care about?

- The Function Call Stack
  - Our Activation Record Instances
  - Also called a Stack Trace
  - Extremely useful for debugging / code tracing

- Any additional information related to the exception

Advantages to Handling Exceptions

- Error detection code is hard to write
- Exception handling shifts the burden onto the runtime / language implementation
- The programmer can focus on the class(es) of exceptions worth handling
- Exception propagation (with stack rewinding!) enables a high-level of reuse for exception handling code

Program Components

Executing Code

The part of the program with usually normal behavior

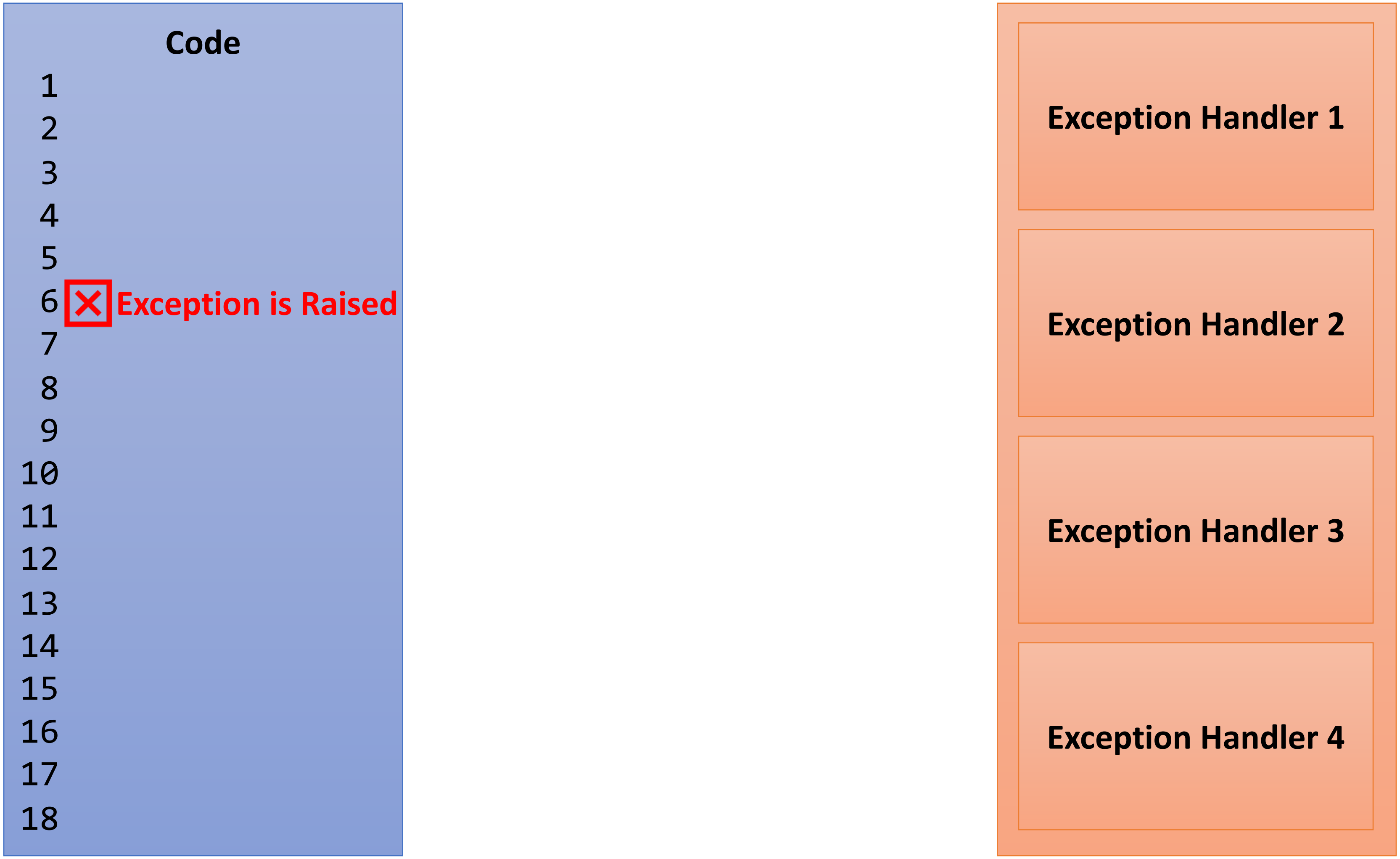
- May introduce an *exception* during execution, usually through some exceptional case:
  - null pointer dereference
  - file not found
  - division by zero

Exception Handlers

The part of the program that either:

1. Recovers the state of the program to a resumable place
2. Informs the user there was a problem
3. Terminates execution of the program
4. Or some combination of the three above

Program Components



Code

1
2
3
4
5
6 ☒ Exception is Raised
7
8
9
10
11
12
13
14
15
16
17
18

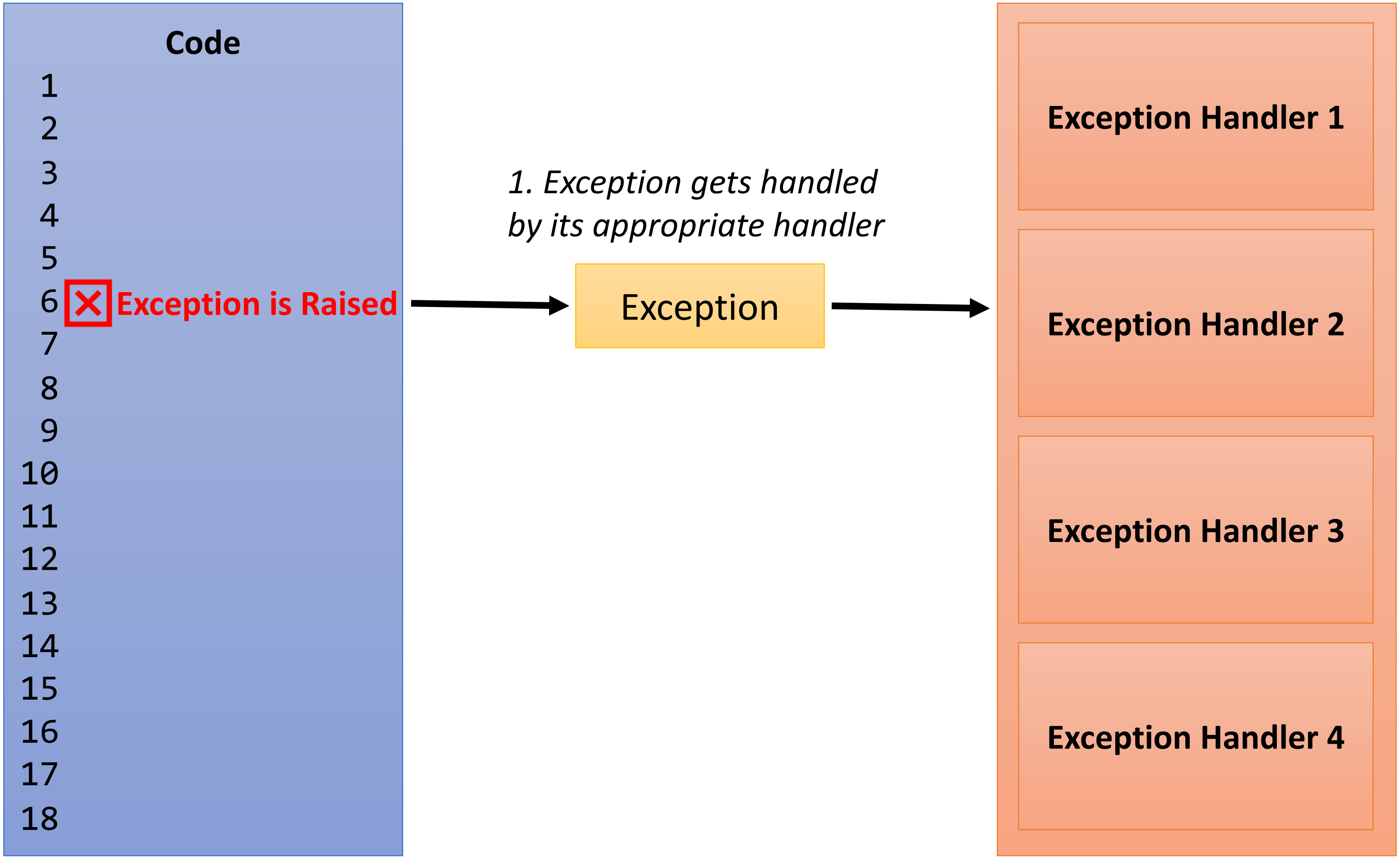
Exception Handler 1

Exception Handler 2

Exception Handler 3

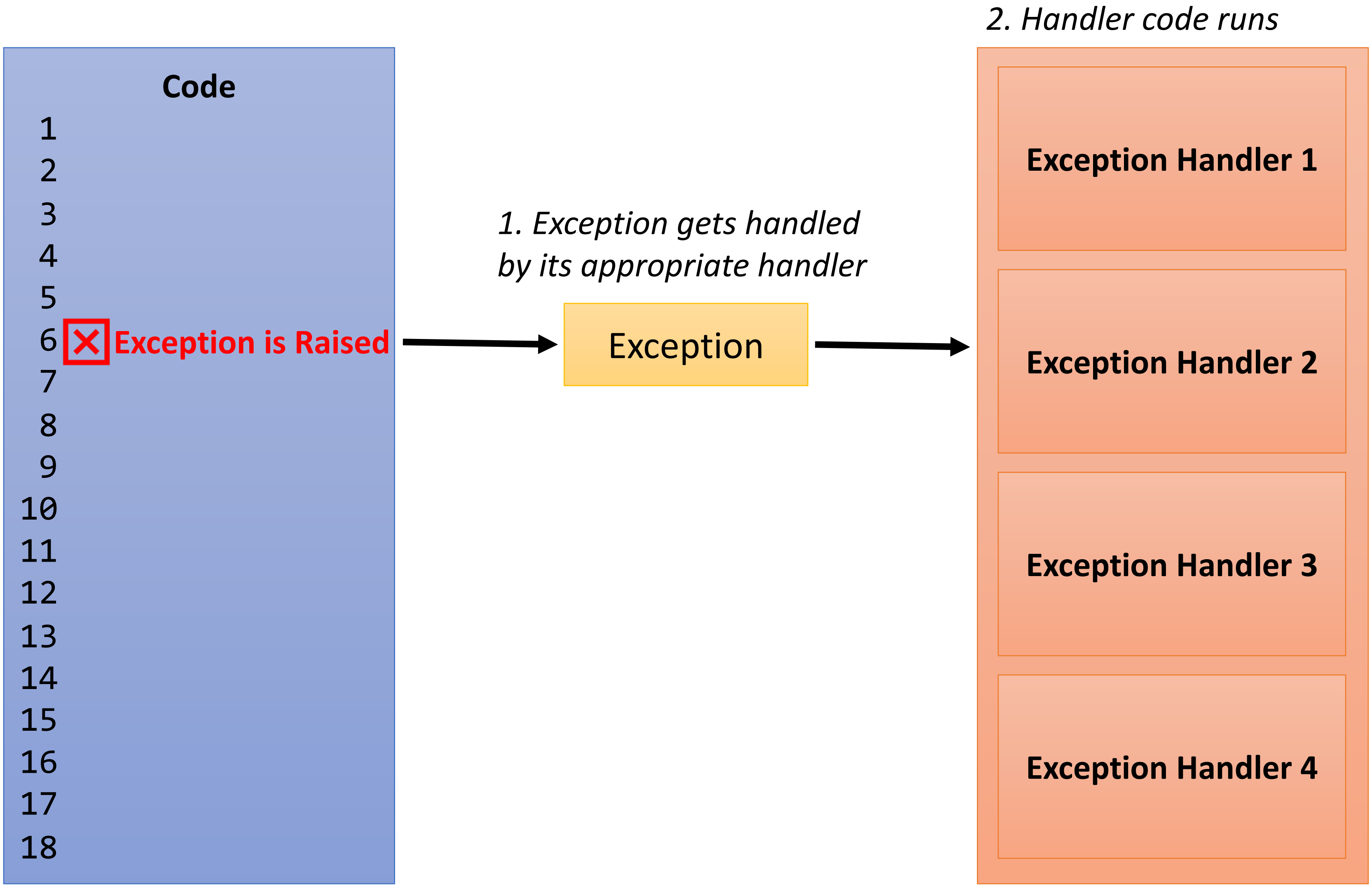
Exception Handler 4

Program Components



1. Exception gets handled by its appropriate handler

Program Components



2. *Handler code runs*

Code

1
2
3
4
5
6 ☒ **Exception is Raised**
7
8
9
10
11
12
13
14
15
16
17
18

1. *Exception gets handled by its appropriate handler*

Exception

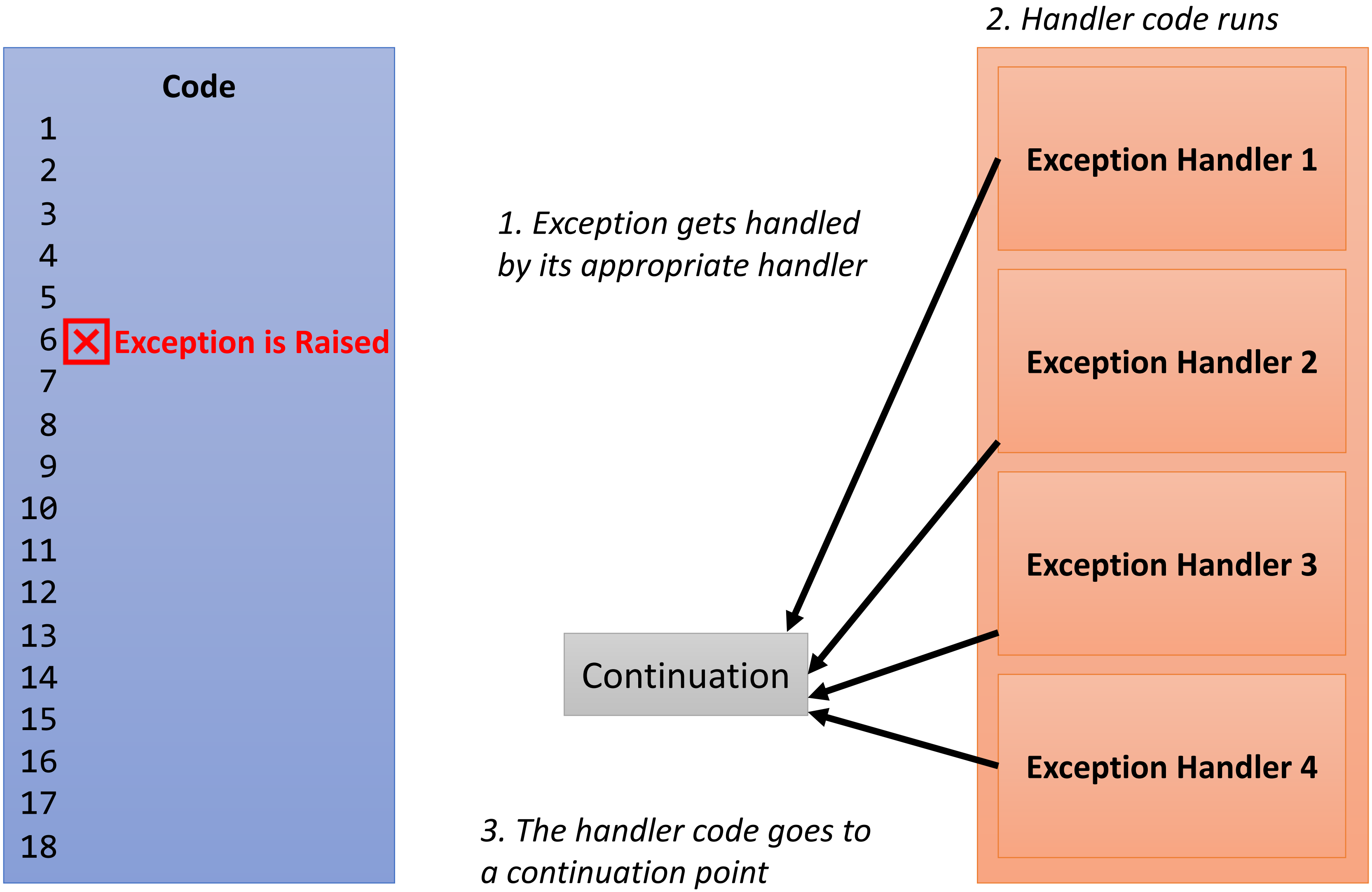
Exception Handler 1

Exception Handler 2

Exception Handler 3

Exception Handler 4

Program Components



Code

1
2
3
4
5
6 ☒ Exception is Raised
7
8
9
10
11
12
13
14
15
16
17
18

1. Exception gets handled by its appropriate handler

2. Handler code runs

Exception Handler 1

Exception Handler 2

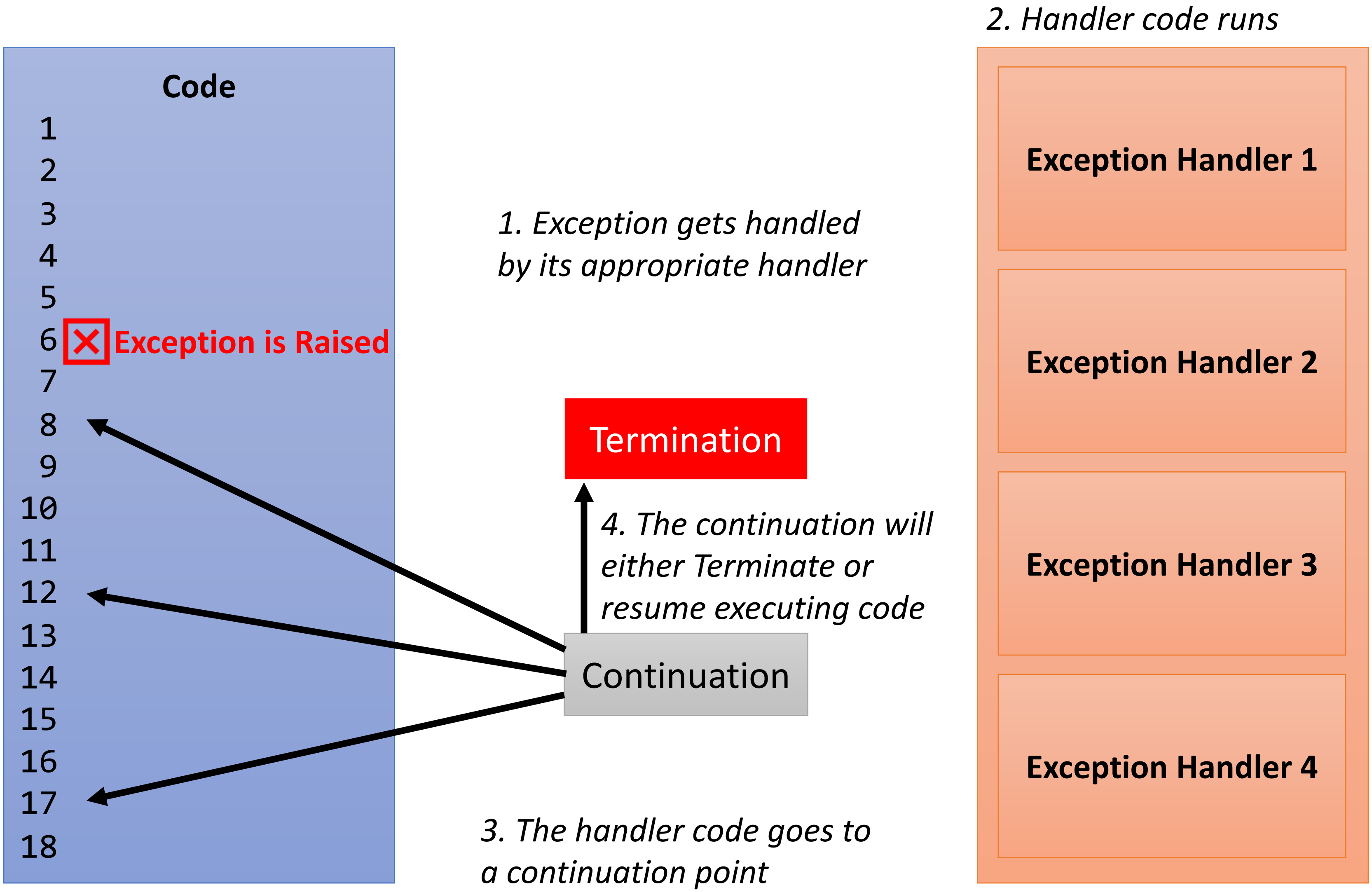
Exception Handler 3

Exception Handler 4

Continuation

3. The handler code goes to a continuation point

Program Components



Code

1
2
3
4
5
6 ☒ Exception is Raised
7
8
9
10
11
12
13
14
15
16
17
18

1. Exception gets handled by its appropriate handler

2. Handler code runs

Exception Handler 1

Exception Handler 2

Exception Handler 3

Exception Handler 4

3. The handler code goes to a continuation point

Continuation

4. The continuation will either Terminate or resume executing code

Termination

Constructs

Creating Exceptional Control Flow

- Our traditional control logic is still present
  - If/then/else, pre-test loops, for-loops, etc.
- Need to introduce special structures that can help manage exceptional control flow

Goals:

- Should be able to mark a region as potentially exception throwing
- Want handler code to co-exist with our executing code
- Should be able to handle different types of exceptions
- Should be able to clean up resources*

Creating Exceptional Control Flow

Should be able to mark a region as potentially exception throwing

Introduce a “try” block

This tells the language that we are going to attempt to run some code, but something exceptional may happen

Creating Exceptional Control Flow

Want handler code to co-exist with our executing code

Introduce a “catch” block

Also called an “exception” block. This tells the language what to do in case an exception occurs. This looks and behaves like an if-then-else

Creating Exceptional Control Flow

Should be able to handle different types of exceptions

Allow multiple “catch” blocks to one try

We should be able to have different handlers run for different exceptions in the same code region

Creating Exceptional Control Flow

Should be able to clean up resources

Introduce a “finally” block

Code that runs regardless of exception or non-exceptional behavior. Used for cleaning up resources guaranteed to be allocated prior to an exception.

Design Issues

- How is an exception instance bound to an exception handler?
- Can/should information about the exception be passed to the handler?
- Where does execution continue, if at all, after an exception handler completes its execution? (continuation vs. resumption)
- Is some form of finalization provided?

Design Issues

- How are user-defined exceptions specified?
- Should there be default exception handlers for programs that do not provide their own?
- Can predefined exceptions be explicitly raised?
- Are hardware-detectable errors treated as exceptions that can be handled?
- Are there any predefined exceptions?
- How can exceptions be disabled, if at all?

Case Studies

C++

C++

```
try {
   // code that could raise an exception
}
catch (formal parameter) {
   // handler code
}
catch (formal parameter) {
   // handler code
}
```

C++ Catch Clauses

Catch-clause that declares a named formal parameter

```
catch (const std::exception& e) { /* */ }
```

Catch-clause that declares an unnamed parameter

```
catch (const std::exception&) { /* */ }
```

Catch-all handler, which is activated for any exception

```
catch (...) { /* */ }
```

C++ Catch Clauses

- **`catch`** is the name of all handlers
- it is an overloaded name, so the formal parameter of each must be unique
- The formal parameter can be used to transfer information to the handler
- The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled

C++ Catch Clauses

```
try {
  f();
} catch (const std::overflow_error& e) {
  // f() throws std::overflow_error
} catch (const std::runtime_error& e) {
  // f() throws std::underflow_error (base class rule)
} catch (const std::exception& e) {
  // f() throws std::logic_error (base class rule)
} catch (...) {
  // f() throws std::string or int or anything else
}
```

C++ Throwing Exceptions

- Exceptions are all raised explicitly by the statement:

  `throw` [*expression*] **;**

- A **`throw`** without an operand can only appear in a handler; when it appears, it simply re-raises the exception, which is then handled elsewhere
- The type of the expression disambiguates the intended handler

C++ Unhandled Exceptions

- An unhandled exception is propagated to the caller of the function in which it is raised
- This propagation continues to the main function
- If no handler is found, the default handler is called
- The default handler, `unexpected`, simply terminates the program; `unexpected` can be redefined by the user

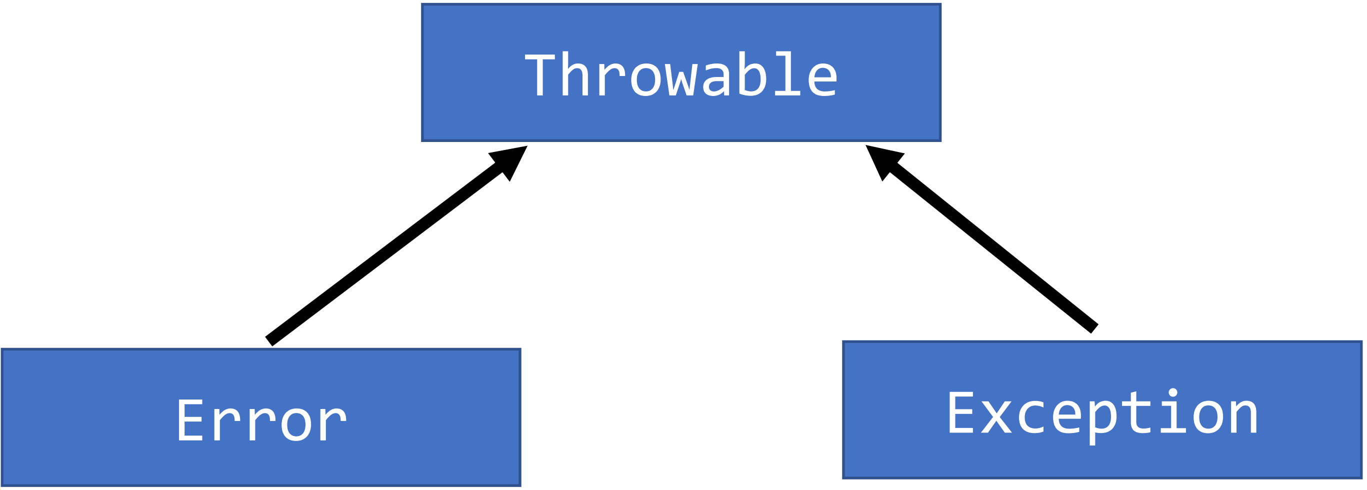
C++ Design Choices

- All exceptions are user-defined
- There are no predefined exceptions
- Exceptions are neither specified nor declared
- Exceptions are not named
- Hardware- and system software-detectable exceptions cannot be handled
- Binding is done via formal parameter types
- Functions can say they do not throw an exception with the noexcept keyword

Java

Java

- Similar philosophy as C++
- Forces exceptions to be *objects*
    - All descendants of ***Throwable***



Java: Error vs. Exception

Error

- Thrown by the Java interpreter for events such as heap overflow
- Never handled by user programs

Exception

- User-defined exceptions are usually subclasses of this
- Has two predefined subclasses:
  - `IOException`
  - `RuntimeException`
    - `ArrayIndexOutOfBoundsException`
    - `NullPointerException`

Java: Exception Handling

- Syntax of try identical to C++
- Exceptions are thrown with throw, but must include the new keyword (explicit dynamic allocation)

```
throw new IllegalArgumentException("Nope")
```

- Handlers are resolved in order. The first to match (or an ancestor to it) will be applied.
- A single handler can be applied to many Exceptions

```
catch (IllegalArgumentException | IllegalStateException)
```

Java: Exception Continuation

- If no handler is found in the **`try`** construct, the search is continued in the nearest enclosing **`try`** construct, etc.
- If no handler is found in the method, the exception is propagated to the method's caller
- If no handler is found (all the way to **`main`**), the program is terminated
- To ensure that all exceptions are caught, a handler can be included in any **`try`** construct that catches all exceptions

Java: Checked Exceptions

- *Checked Exceptions* thrown within a method must be either
  1. catched (or handled) within the method or
  2. Listed explicitly in the throws clause of a method

```
public static File load(String name)
throws FileNotFoundException
```

- Error, RuntimeException, and their descendants are all considered *Unchecked Exceptions*
- Everything else is considered a *Checked Exception*
- The throws clause is part of the function signature

Java: Throws Clause (on Function)

- The `throws` clause is part of the function signature
- A method cannot declare more exceptions in its throws clause than the method it overrides
- A method that calls a method that lists a particular checked exception in its throws clause has three alternatives for dealing with that exception:
  - Catch and handle the exception
    Catch the exception and throw an exception that is listed in its own throws clause
  - Declare it in its throws clause and do not handle it

Java: Finally Clause

- The `finally` clause can appear at the end of a `try`
- Purpose: To run code regardless of what happens in the try construct (or handlers that don't throw)

```java
try {
    read = new Scanner(s);
    File file = new File(path);
    // use file that does not exist
} catch (Exception e) {
    // report error
} finally {
    read.close();
}
```

Java: Assertions

- There is one more class of exception-enabling constructs present in the Java programming language

```
assert condition
assert condition: expression
```

- When evaluated to true, nothing happens
- When evaluated to false, `AssertionError` is raised
- Can be disabled during runtime without recompiling

Java: Design Choices

- The types of exceptions present a clean hierarchy which is easily extendable
- The `throws` clause attached to the function signature helps understand the exceptional control flow contract in large systems
- The `finally` clause provides additional flexibility in response to potential resource leaks
- The Java language implementation raises exceptions that can be caught by user (client) code

Python

Python

- Like Java, Exceptions are objects
  - `BaseException` abstract base class
- All predefined and user-defined exceptions are derived from `Exception`
- Predefined subclasses of `Exception`:
  - `ArithmeticError`
    - `OverflowError`
    - `ZeroDivisonError`
    - `FloatingPointError`
  - `LookupError`
    - `IndexError`
    - `KeyError`

Python Example

```
try:
    # execute code
except Exception1:
    # Handler for Exception1
except Exception2:
    # Handler for Exception2
else:
    # execute when no exception is raised
finally:
    # execute no matter what
```

Python: Exception Handling

- Handlers handle exceptions raised with the exact name plus all subclasses
- Unhandled exceptions get propagated to the nearest enclosing `try` block.
- If no handler for the exception is found, the default handler is called
- Exceptions can be “raised” with the `raise` keyword
  - `raise IndexError`
- An instance of the exception raised can be retrieved
  - `except Exception as ex_obj`

Python: Exception Raising

- Exceptions can be “raised” with the `raise` keyword
  - `raise IndexError`

- The assert statement is similar to Java’s
  - `assert` ***test***, *data*
  - Tests the Boolean expression, ***test***
  - If the test fails, send the second parameter, *data*, to the Exception object to be raised

OCaml

OCaml

- Exceptions belong to the type exn
  - exn is an extensible sum type

- The biggest issue with exceptions is that they do not appear in types.

- Must rely on documentation to see that a function may throw an exception

OCaml: Defining Exceptions

```
exception Foo of string
(* Syntax: exception Tag [of inner] *)

let i_will_fail () =
  raise (Foo "ohnoes!")

(* creating a new instance is as easy as
  creating a discriminated union value *)
```

OCaml: Handling Exceptions

```ocaml
let safe_inverse n =
  try Some (1 / n)
  with Division_by_zero -> None

let safe_list_find p l =
  try Some (List.find p l)
  with Not_found -> None
```

OCaml: Handling Exceptions

`try` *`expr`*

- *`expr`* is any OCaml expression

`with` *`exn_match`*

- *`exn_match`* is a special pattern match
  - the exception type must be the type matched
  - the value result of the pattern match matching the expression type from the `try`

OCaml: Printing Exceptions

```
let notify_user f =
  try f()
  with e -> (* implicit type *)
    let msg = Printexc.to_string e
    and stack = Printexc.get_backtrace ()
    in Printf.eprintf
          "there was an error: %s%s\n"
          msg stack;
       raise e
```

OCaml: User-Defined Exceptions

```
exception Foo of int

let () =
  Printexc.register_printer
    (function
     | Foo i -> Some (Printf.sprintf "Foo(%d)" i)
     | _ -> None (* for other exceptions *) )
```