Programming Paradigms

Control Abstraction (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2022

Overview

- Calling Sequences
- Parameter Passing
- Exception Handling 
- Coroutines
- Promises, Async, and Await

Quiz: Exceptions

What does the following Java code print?

```
try {
    try {
        Object obj = null;
        obj.equals(obj);
    } catch (IllegalStateException e) {
        System.out.println("Caught it!");
    } catch (NullPointerException e) {
        throw new RuntimeException(e);
    }
} catch (NullPointerException e) {
    System.out.println("Also caught it!");
} finally {
    System.out.println("Finally!");
}
```

Quiz: Exceptions

What does the following Java code print?

```
try {
    try {
        Object obj = null;
        obj.equals(obj);
    } catch (IllegalStateException e) {
        System.out.println("Caught it!");
    } catch (NullPointerException e) {
        throw new RuntimeException(e);
    }
} catch (NullPointerException e) {
    System.out.println("Also caught it!");
} finally {
    System.out.println("Finally!");
}
```

Result:

Finally!

Exception in ...

Quiz: Exceptions

What does the following Java code print?

```
try {
    try {
        Object obj = null;
        obj.equals(obj);
    } catch (IllegalStateException e) {
        System.out.println("Caught it!");
    } catch (NullPointerException e) {
        throw new RuntimeException(e);
    }
} catch (NullPointerException e) {
    System.out.println("Also caught it!");
} finally {
    System.out.println("Finally!");
}
```

Throws a NullPointerException

Quiz: Exceptions

What does the following Java code print?

```
try {
    try {
        Object obj = null;
        obj.equals(obj);
    } catch (IllegalStateException e) {
        System.out.println("Caught it!");
    } catch (NullPointerException e) {
        throw new RuntimeException(e);
    }
} catch (NullPointerException e) {
    System.out.println("Also caught it!");
} finally {
    System.out.println("Finally!");
}
```

Wrong exception type:

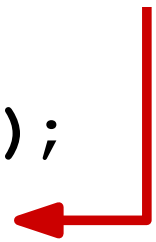
Nothing caught here.

Quiz: Exceptions

What does the following Java code print?

```
try {
    try {
        Object obj = null;
        obj.equals(obj);
    } catch (IllegalStateException e) {
        System.out.println("Caught it!");
    } catch (NullPointerException e) {
        throw new RuntimeException(e);
    }
} catch (NullPointerException e) {
    System.out.println("Also caught it!");
} finally {
    System.out.println("Finally!");
}
```

Catches `e` and wraps it into another exception

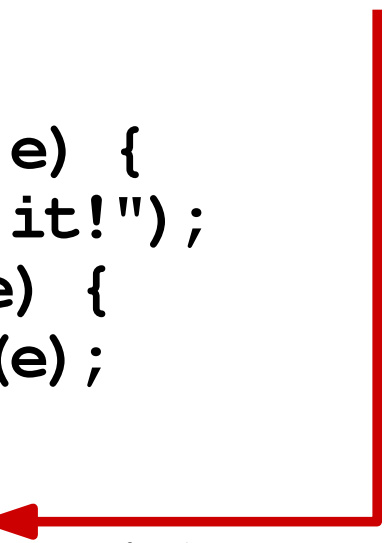


Quiz: Exceptions

What does the following Java code print?

Not a `NullPointerException` anymore: Nothing caught here

```
try {
    try {
        Object obj = null;
        obj.equals(obj);
    } catch (IllegalStateException e) {
        System.out.println("Caught it!");
    } catch (NullPointerException e) {
        throw new RuntimeException(e);
    }
} catch (NullPointerException e) {
    System.out.println("Also caught it!");
} finally {
    System.out.println("Finally!");
}
```



Quiz: Exceptions

What does the following Java code print?

```
try {
    try {
        Object obj = null;
        obj.equals(obj);
    } catch (IllegalStateException e) {
        System.out.println("Caught it!");
    } catch (NullPointerException e) {
        throw new RuntimeException(e);
    }
} catch (NullPointerException e) {
    System.out.println("Also caught it!");
} finally {
    System.out.println("Finally!");
}
```

`finally` blocks are always executed

Exceptions

- **Exception**: Unusual condition during execution that cannot be easily handled in local context
- Raising an exception **diverges from normal control flow**
- **Exception handler**: Code executed when an exception occurs

When Do Exceptions Occur?

- **Implicitly thrown by language implementation**
  - Runtime errors, e.g., division by zero
- **Explicitly thrown by program**
  - Illegal or unexpected program state, e.g., combination of flags that should never occur
- **Don't use exceptions to encode "normal" control flow**

Alternatives to Exceptions

In PL without exceptions, other options include:

- "Invent" a return value
  - E.g., empty string if cannot read from file
- Encode status in return value
  - E.g., as an integer error code
- Caller passes a closure with error-handling routine
  - E.g., "error-first" callback on Node.js

Syntax of Exceptions

Most common in modern PLs:

Try-catch blocks

- Handler is lexically bound to block of code
- Example (C++):

```
try {
  // ...
  if (something_unexpected}
    throw my_error("oops");
  // ...
} catch (my_error e) {
  // handle exception
}
```

Syntax of Exceptions

Most common in modern PLs:

Try-catch blocks

- Handler is lexically bound to block of code
- Example (C++):

```
try {
  // ...
  if (something_unexpected)
    throw my_error("oops");
  // ...
} catch (my_error e) {
  // handle exception
}
```

Handler for specific type of exception

Nested Try Blocks

- If exception thrown, control passed to inner-most matching handler

```
try {
  try {
    // ...
    // code that may throw exception
    // ...
  } catch (some_other_error e) {
    // handle some_other_error
  }
} catch (my_error e) {
  // handle my_error
}
```

Nested Try Blocks

- If exception thrown, control passed to inner-most matching handler

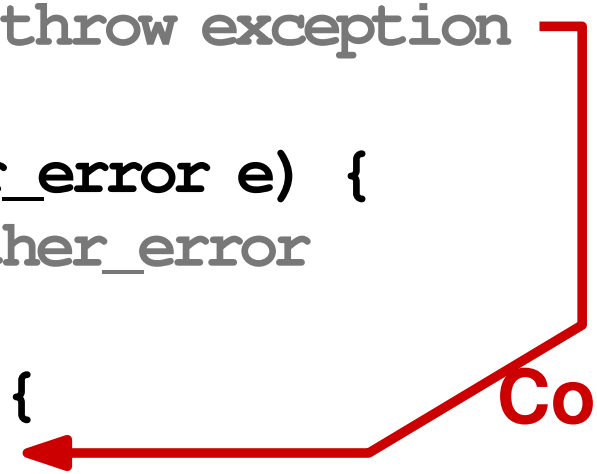
```
try {
  try {
    // ...
    // code that may throw exception
    // ...
  } catch (some_other_error e) {
    // handle some_other_error
  }
} catch (my_error e) {
  // handle my_error
}
```

Control flow if `some_other_error` thrown

Nested Try Blocks

- If exception thrown, control passed to inner-most matching handler

```
try {
  try {
    // ...
    // code that may throw exception
    // ...
  } catch (some_other_error e) {
    // handle some_other_error
  }
} catch (my_error e) {
  // handle my_error
}
```



Control flow if `my_error` thrown

Lists of Handlers

- If different exceptions may be thrown in same block, use list of handlers

```
try {
  // code that may throw exception
} catch (end_of_file e) {
  // handle end of file
} catch (io_error  e) {
  // handle I/O errors
} catch (...) {
  // handles any not previously handled exception
}
```

Lists of Handlers

- If different exceptions may be thrown in same block, use list of handlers

```
try {
  // code that may throw exception
} catch (end_of_file e) {
  // handle end of file
} catch (io_error  e) {
  // handle I/O errors
} catch (...) {
  // handles any not previously handled exception
}
```

C++ syntax for "catch all"

Propagation Outside Subroutine

What if no matching handler in current subroutine?

- Immediately return and re-raise exception at call site
- May propagate until main routine
  - Unwinds stack without finishing routines
- If not handled at all, terminate program

Defining Exceptions

Mechanisms vary across PLs

- Subtype of particular class
  - E.g., in Java, subtypes of `Exception`
- Special kinds of objects (akin to constants, types, variables)
  - E.g., in Modula-3:

    ```
    EXCEPTION empty_queue
    ```
- Any value that exists in the PL
  - E.g., JavaScript:

    `throw 42;` or `throw "Expected a number";`

How to Handle Exceptions?

- **Recover and continue execution**
  - E.g., if out of memory, allocate more memory
- **Clean up locally allocated resources and re-raise exception to be handled elsewhere**
  - E.g., close opened files
- **Print error message and terminate program**

How to Handle Exceptions?

- **Recover and continue execution**
  - E.g., if out of memory, allocate more memory
- **Clean up locally allocated resources and re-raise exception to be handled elsewhere**
  - E.g., close opened files
- **Print error message and terminate program**

Do not just swallow exceptions!

Declaring Exceptions

In some PLs, possibly thrown exceptions are part of the subroutine header

- Must declare every exception, e.g., Modula-3
- Declaring exceptions is optional, e.g., C++
- Checked vs. unchecked exceptions, e.g., Java
  - Must declare checked exceptions
  - Optional for unchecked exceptions

Cleanup Operations

- **`finally` clause: Executed whenever control leaves the current block**
  - When exception is thrown
  - Also when no exception thrown
- **Use to clean up local state**
  - E.g., release resources

Quiz: Exceptions (2)

What does this Python code print?

```
def f():
    try:
        raise "oops"
    except:
        print("a")
    finally:
        print("b")

def g():
    try:
        print("c")
    except:
        print("d")
    finally:
        f()
        print("e")

g()
```

Quiz: Exceptions (2)

What does this Python code print?

Result:

c

a

b

e

```
def f():
    try:
        raise "oops"
    except:
        print("a")
    finally:
        print("b")

def g():
    try:
        print("c")
    except:
        print("d")
    finally:
        f()
        print("e")

g()
```

Overview

- Calling Sequences
- Parameter Passing
- Exception Handling
- Coroutines 
- Promises, Async, and Await

Coroutines

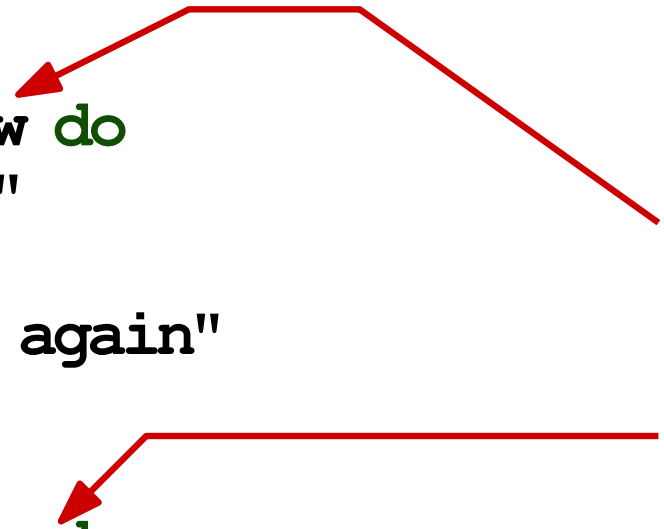
- **Control abstraction that allows for**
  - suspending execution
  - resuming where it was suspended
- **For implementing non-preemptive multi-tasking**

Example: Fibers in Ruby

```
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```

Example: Fibers in Ruby

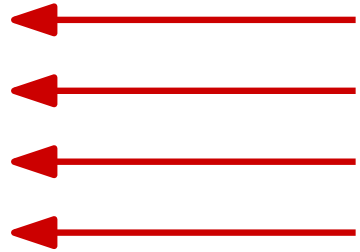
```
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```



Creates a coroutine (“fiber”)

Example: Fibers in Ruby

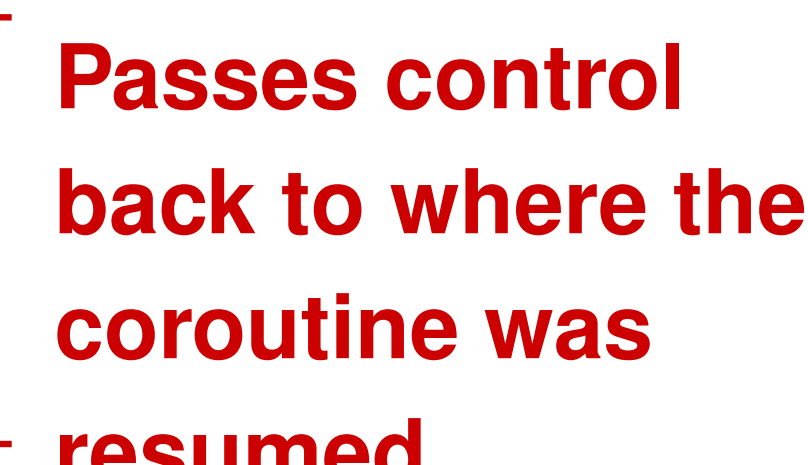
```
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```



Continues to run a coroutine from where it last stopped

Example: Fibers in Ruby

```
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```



Example: Fibers in Ruby

```
fiber1 = Fiber.new do
    puts "Fiber 1"
    Fiber.yield
    puts "Fiber 1 again"
end

fiber2 = Fiber.new do
    puts "Fiber 2"
    Fiber.yield
    puts "Fiber 2 again"
end

fiber1.resume
fiber2.resume
fiber2.resume
fiber1.resume
```

Prints:

Fiber 1

Fiber 2

Fiber 2 again

Fiber 1 again

Coroutines vs. Threads

- Explicit transfer of control (non-preemptive)
- Only one coroutines runs at a time

- Control flow transfered implicitly and preemptively
- Multiple threads may run concurrently

Coroutines vs. Continuations

- Changes every time it runs
- Old program counter saved when transfering to another coroutines
- When transfering back, continue where we left off

- Once created, doesn't change
- When invoking, old program counter is lost
- Multiple jumps to same continuation always start at some position

Coroutines vs. Continuations

- Changes every time it runs
- Old program counter saved when transfering to another coroutines
- When transfering back, continue where we left off

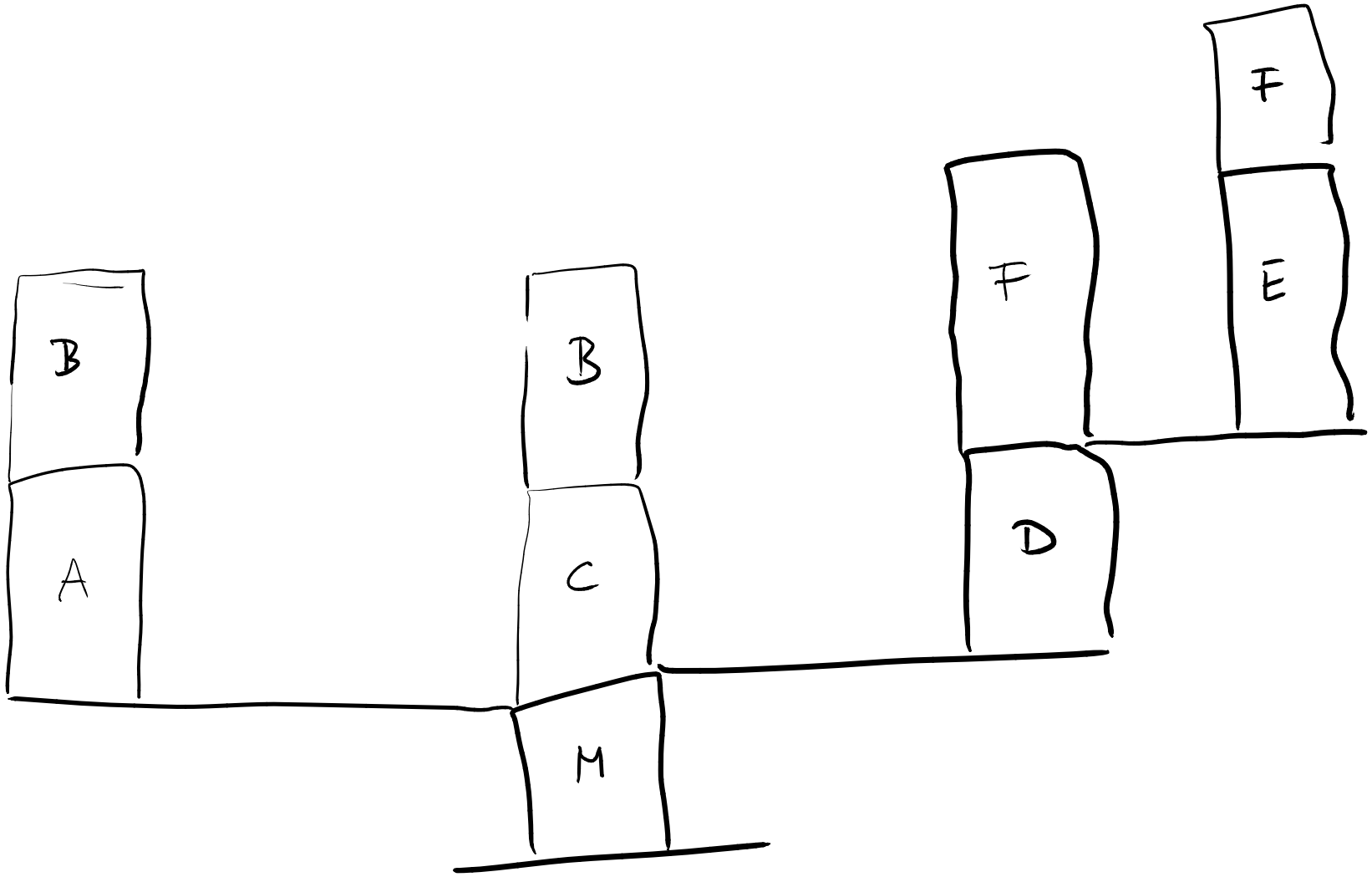
- Once created, doesn't change
- When invoking, old program counter is lost
- Multiple jumps to same continuation always start at some position

Both: Represented by a closure
(= code address + referencing environment)

Stack Allocation

- **Coroutines may call subroutines and create other coroutines**
- **Each coroutine has its own function stack**
  - Second stack created when a routine creates a coroutine
- **Repeated creation of coroutines: "Cactus stack"**

Cactus Stack



Coroutines in Popular PLs

- **Natively** supported, e.g., in Ruby and Go
- Available as **libraries**, e.g., for Java, C#, JavaScript, Kotlin
- **Specialized variants**, e.g., in Python (generators)

Overview

- Calling Sequences
- Parameter Passing
- Exception Handling
- Coroutines
- Promises, Async, and Await 

Motivation for Asynchrony

- **Parts of a program may take very long**
  - File I/O
  - Network I/O
  - Waiting for user input
- **Continue with rest of program until long-running parts are finished**

Expressing Asynchrony

- **Event-driven programming**
  - Register callbacks to invoke once finished
- **Promises (aka futures)**
  - Object to represent a not yet computed value
- **Async and await**
  - Syntactic sugar to ease programming with promises

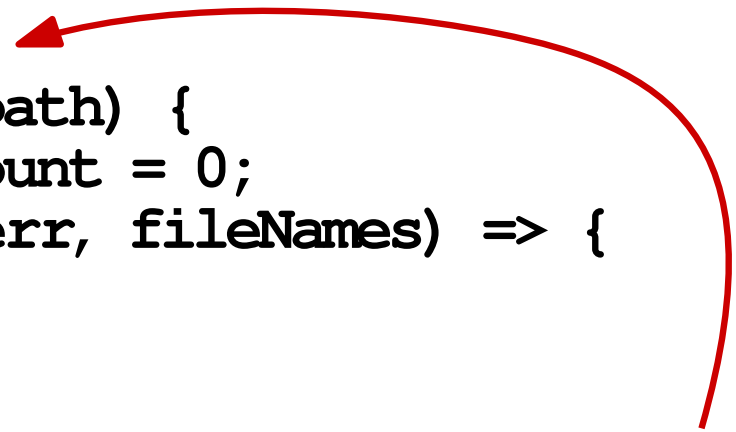
Event-Driven Programming

Minimal example (JavaScript):

```
longRunning(someArg, (err, result) => {
  if (err == null)
    // handle error
  else
    // use result
})
```

Example: Sum of File Sizes

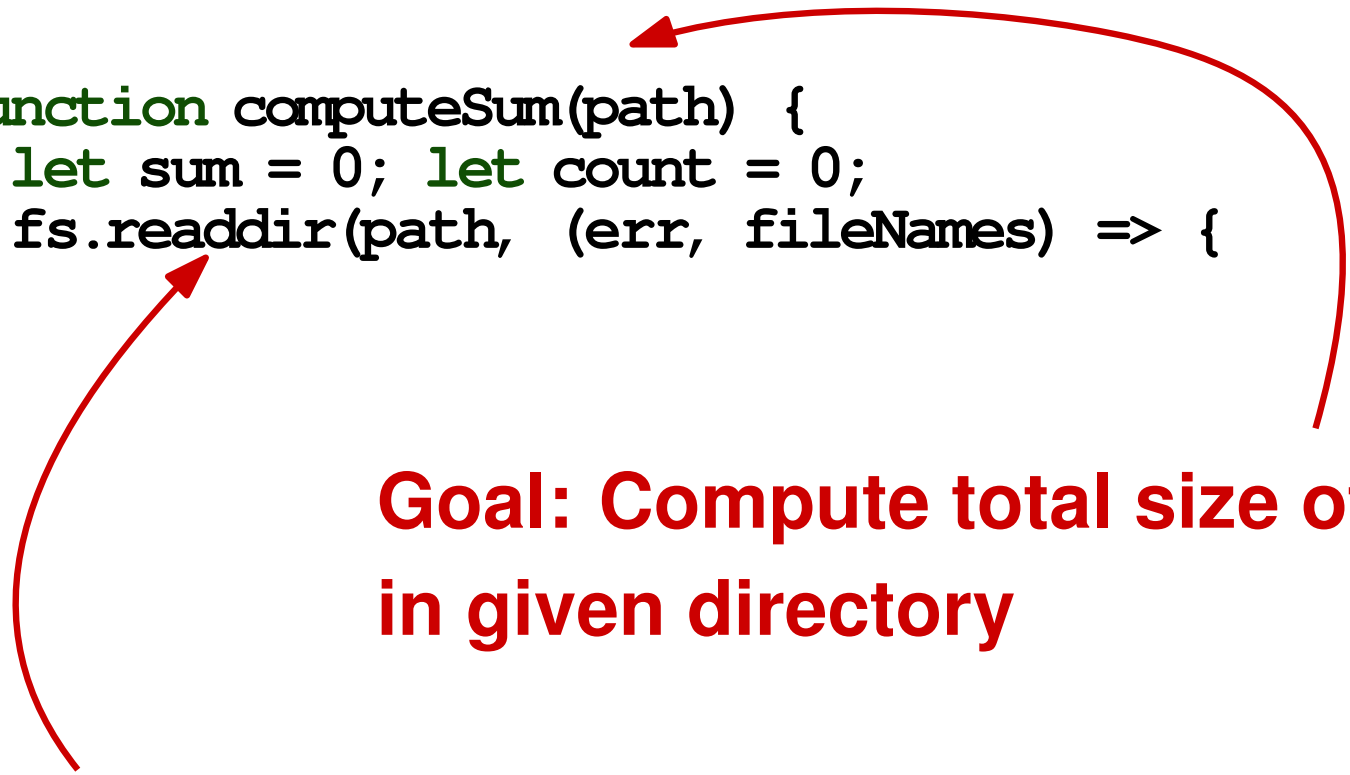
```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {

  });
}
```



Goal: Compute total size of all files in given directory

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {

  });
}
```



Goal: Compute total size of all files in given directory

Read files in directory and invoke callback once done

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {
    if (err === null){
```

Handle possible errors while during file I/O

```
    } else
      console.log("I/O error: " + err);
  });
}
```

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {
    if (err === null){
      for (let fileName of fileNames) {
        fs.stat(fileName, (err, fileInfo) => {

        });
      }
    } else
      console.log("I/O error: " + err);
  });
}
```

Get file information (incl. size) for each file in the directory

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {
    if (err === null){
      for (let fileName of fileNames) {
        fs.stat(fileName, (err, fileInfo) => {
          if (err === null) {
            } else
              console.log("I/O error: " + err);
          });
        }
    } else
      console.log("I/O error: " + err);
  });
}
```

Example: Sum of File Sizes

```
function computeSum(path) {
  let sum = 0; let count = 0;
  fs.readdir(path, (err, fileNames) => {
    if (err === null){
      for (let fileName of fileNames) {
        fs.stat(fileName, (err, fileInfo) => {
          if (err === null) {
              sum += fileInfo.size;
              count++;
              if (count == fileNames.length) {
                console.log(sum);
              }
          } else
            console.log("I/O error: " + err);
        });
    } else
      console.log("I/O error: " + err);
  });
}
```

Synchronization: Ensure to write sum once callbacks for all files invoked

Problems

- Deeply nested control flow: "Callback hell"
- Error-handling scattered throughout code
- Needs explicit synchronization when depending on multiple asynchronous computations

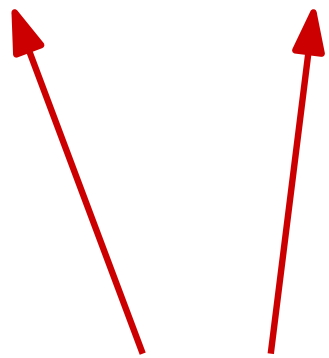
Promises

- **Object that represents result of asynchronous computation**
- **Always in one of three states**
  - Pending
  - Resolved
  - Rejected

  } **Settled**
- **Once settled, state doesn't changed anymore**

Minimal Example

```
// 1) Create a promise
let p = new Promise((resolve, reject) => {
  if (...)
    resolve(someValue);
  else
    reject(someError);
});
```



Functions to call for resolving/rejecting the promise

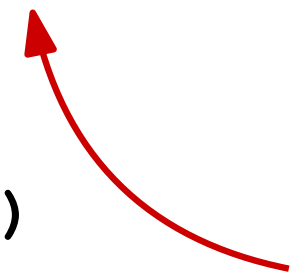
Minimal Example

```
// 1) Create a promise
let p = new Promise((resolve, reject) => {
  if (...)
    resolve(someValue);
  else
    reject(someError);
});

// 2) Use the promise
p.then((x) => {
  // use resulting value
}).catch((e) => {
  // handle error
};
```

Register *reaction*
invoked when promise
is resolved/rejected

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {


  })


}
```



Now using the promise version of `fs` APIs, which return promises

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {
    const promises = fileNames.map((fn) => fs.stat(fn));
    // wait for all of them to be resolved
    return Promise.all(promises);
  })
}
```

Each call returns a promise

Returns a single promise
once all given promises
are resolved

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {
    const promises = fileNames.map((fn) => fs.stat(fn));
    // wait for all of them to be resolved
    return Promise.all(promises);
  }).then((fileInfos) => {

  })

}
```

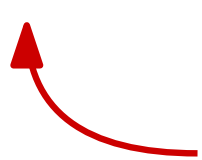
Chain multiple promises:
Reactions registered with `then`
are executed sequentially

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {
    const promises = fileNames.map((fn) => fs.stat(fn));
    // wait for all of them to be resolved
    return Promise.all(promises);
  }).then((fileInfos) => {
    // compute sum
    const sum = fileInfos.reduce((acc, val) =>
      { return acc + val.size; }, 0);
    console.log(sum);
  })


}
```

Example: Sum of File Sizes

```
function computeSum(path) {
  fs.readdir(path).then((fileNames) => {
    const promises = fileNames.map((fn) => fs.stat(fn));
    // wait for all of them to be resolved
    return Promise.all(promises);
  }).then((fileInfos) => {
    // compute sum
    const sum = fileInfos.reduce((acc, val) =>
      { return acc + val.size; }, 0);
    console.log(sum);
  }).catch((e) => {
    console.log("error: " + e);
  });
}
```



Handles errors in any previous promises in the chain

Pros and Cons

- **Benefits over event-driven code**
  - Control flow now easier to understand
  - Explicit synchronization using `Promise.all`
  - All error handling in one place
- **Still suboptimal:**
  - Somewhat verbose syntax due to higher-order functions

Async and Await

- **Label function as `async` if it performs asynchronous computation**
  - Returns a promise
  - May `await` other asynchronous computations
  - No need for higher-order `then` and `catch` functions
  - Error handling using standard `try` and `catch`

Minimal Example

```
async function longRunning() {
  return someValue;
}
```

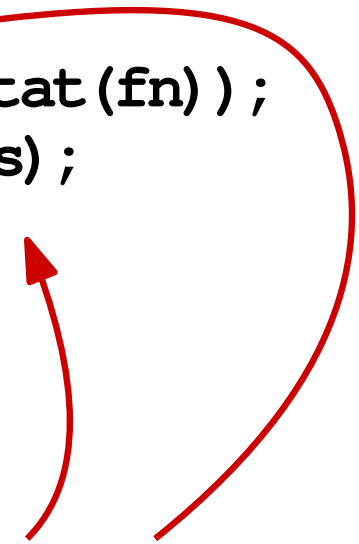
Returns a promise

```
// in some other async function:
let result = await longRunning();
```

Waits for the promise to resolve

Example: Sum of File Sizes

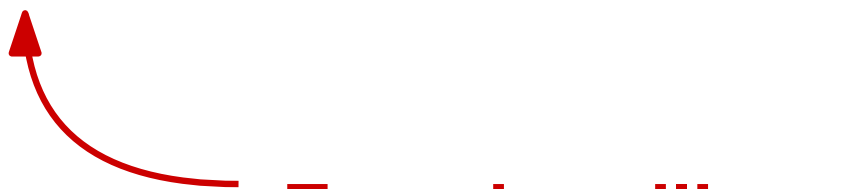
```
async function computeSum(path) {

    const fileNames = await fs.readdir(path);
    const promises = fileNames.map((fn) => fs.stat(fn));
    const fileInfos = await Promise.all(promises);
    const sum = fileInfos.reduce((acc, val) =>
      { return acc + val.size; }, 0);
    console.log(sum);



}
```



Looks like sequential control flow, but execution isn't blocked on await expression

Example: Sum of File Sizes

```
async function computeSum(path) {
  try {
    const fileNames = await fs.readdir(path);
    const promises = fileNames.map((fn) => fs.stat(fn));
    const fileInfos = await Promise.all(promises);
    const sum = fileInfos.reduce((acc, val) =>
      { return acc + val.size; }, 0);
    console.log(sum);
  } catch(e) {
    console.log("error: " + e);
  }
}
```



Error handling via standard `try` and `catch`

Quiz: Promises, Async, and Await

Which of the following statements is true?

- The value represented by a promise may never materialize.
- The semantics of `async` and `await` can be explained in terms of promises.
- All `await` expressions are evaluated in parallel.
- Chained promises are executed one after another.

Quiz: Promises, Async, and Await

Which of the following statements is true?

- The value represented by a promise may never materialize.
- The semantics of `async` and `await` can be explained in terms of promises.
- ~~All `await` expressions are evaluated in parallel.~~
- Chained promises are executed one after another.

Overview

- Calling Sequences
- Parameter Passing
- Exception Handling
- Coroutines
- Promises, Async, and Await ✔