Programming Paradigms

Lecture 13:

Subroutines and Control Abstraction (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2019/2020

Wake-up Exercise

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("Caught it, too.");
} finally {
    System.out.println("Finally here.");
}
```

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

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("Caught it, too.");
} finally {
    System.out.println("Finally here.");
}
```

Result:

Finally here.

Exception in ...

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

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("Caught it, too.");
} finally {
    System.out.println("Finally here.");
}
```

Throws a NullPointerException

https://ilias3.uni-stuttgart.de/vote/0ZT9

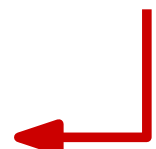
Wake-up Exercise

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("Caught it, too.");
} finally {
    System.out.println("Finally here.");
}
```

Wrong exception type:

Nothing caught here.



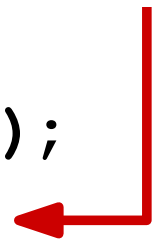
https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

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("Caught it, too.");
} finally {
    System.out.println("Finally here.");
}
```

Catches `e` and wraps it into another exception



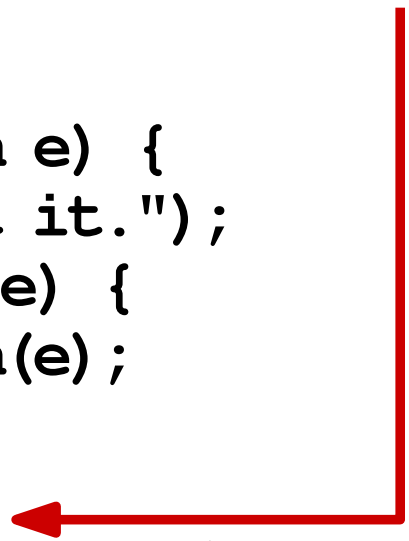
https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

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("Caught it, too.");
} finally {
    System.out.println("Finally here.");
}
```



https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

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("Caught it, too.");
} finally {
    System.out.println("Finally here.");
}
```

`finally` blocks are always executed

https://ilias3.uni-stuttgart.de/vote/0ZT9

Overview

- Calling Sequences
- Parameter Passing
- Exception Handling 
- Coroutines
- Events

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, three other options

- "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

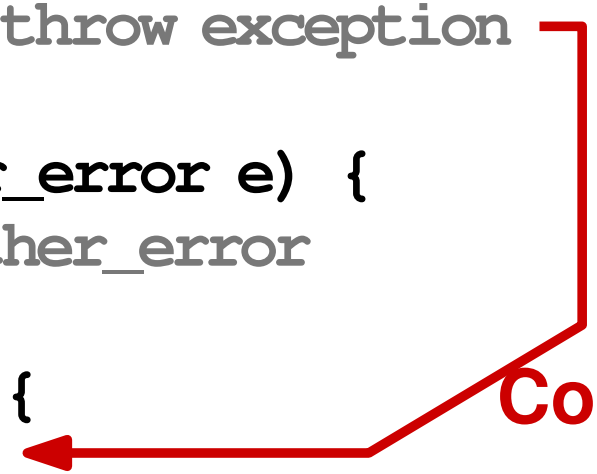
```
try {
  try {
    // ...
    // code that may throw exception
    // ...
  } catch (some_other_error e) {
    // handle some_other_error
  }
} catch (my_error e) {
  // handle my_error
}
```

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 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 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 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 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

What does this Python code print?

```
def f():
    try:
        print("a")
    except:
        print("b")
    finally:
        g()
        print("c")

def g():
    try:
        raise "oops"
    except:
        print("d")
    finally:
        print("e")

f()
```

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: Exceptions

What does this Python code print?

```
def f():
    try:
        print("a")
    except:
        print("b")
    finally:
        g()
        print("c")

def g():
    try:
        raise "oops"
    except:
        print("d")
    finally:
        print("e")

f()
```

Result:

a

d

e

c

https://ilias3.uni-stuttgart.de/vote/0ZT9

Overview

- Calling Sequences
- Parameter Passing
- Exception Handling
- Coroutines 
- Events

Coroutines

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

Coroutines (Pseudo Code)

```
us, cfs : coroutine

coroutine check_file_system ()
    detach // create coroutine &
           // return reference to caller

    for all files
        ...
        transfer (us)
        ...

main:
    us := new update_screen()
    cfs := new check_file_system ()
    transfer (us)
```

```
coroutine update_screen ()
    detach
    loop
        ....
        transfer (cfs)
```

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)

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

Stack Allocation

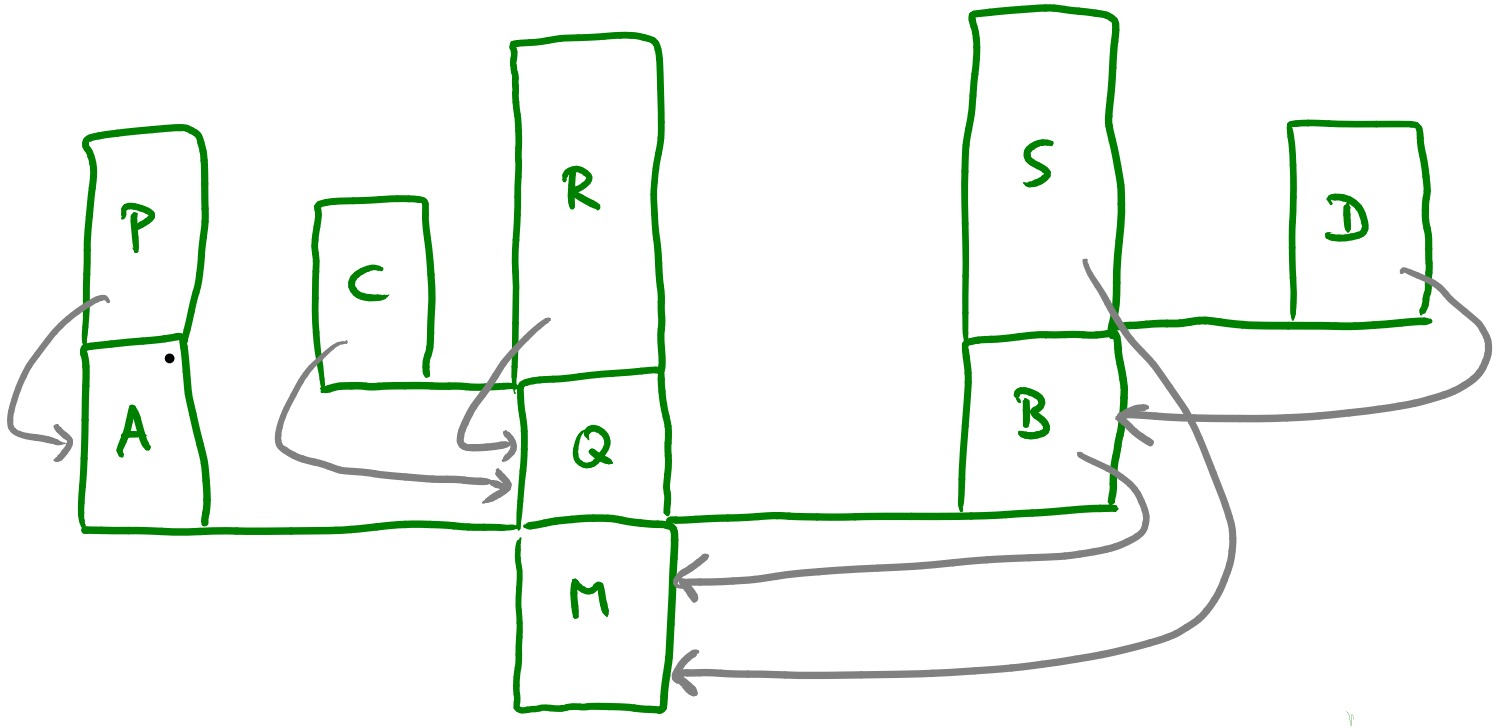
- **Coroutines may call subroutines and create other coroutines**
- **Each coroutine has its own function stack**
  - Second stack created when a routine creates a coroutine
- **Repeated creation of coroutines: "Cactus stack"**

Example: Cactus Stack

Nesting of routine declarations:

- A
  - P
- M
  - B
    - D
  - S
- Q
  - C
  - R

→ .. static links



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
- Events 

Events

- **Event: Something a program needs to react to at an unpredictable time**
  - GUI events, e.g., mouse clicks
  - Asynchronous I/O
- **Event handler: Routine called when a specific kind of event happens**
  - Sequential handlers
  - Thread-based handlers

Sequential Handlers

- **Handle event in main thread of execution**
- **E.g., OS-level interrupt handlers**
  - Register handler for specific interrupt condition
  - Triggered at hardware level
  - OS transfers control to handler and restores state afterwards

Example: UNIX Signaling

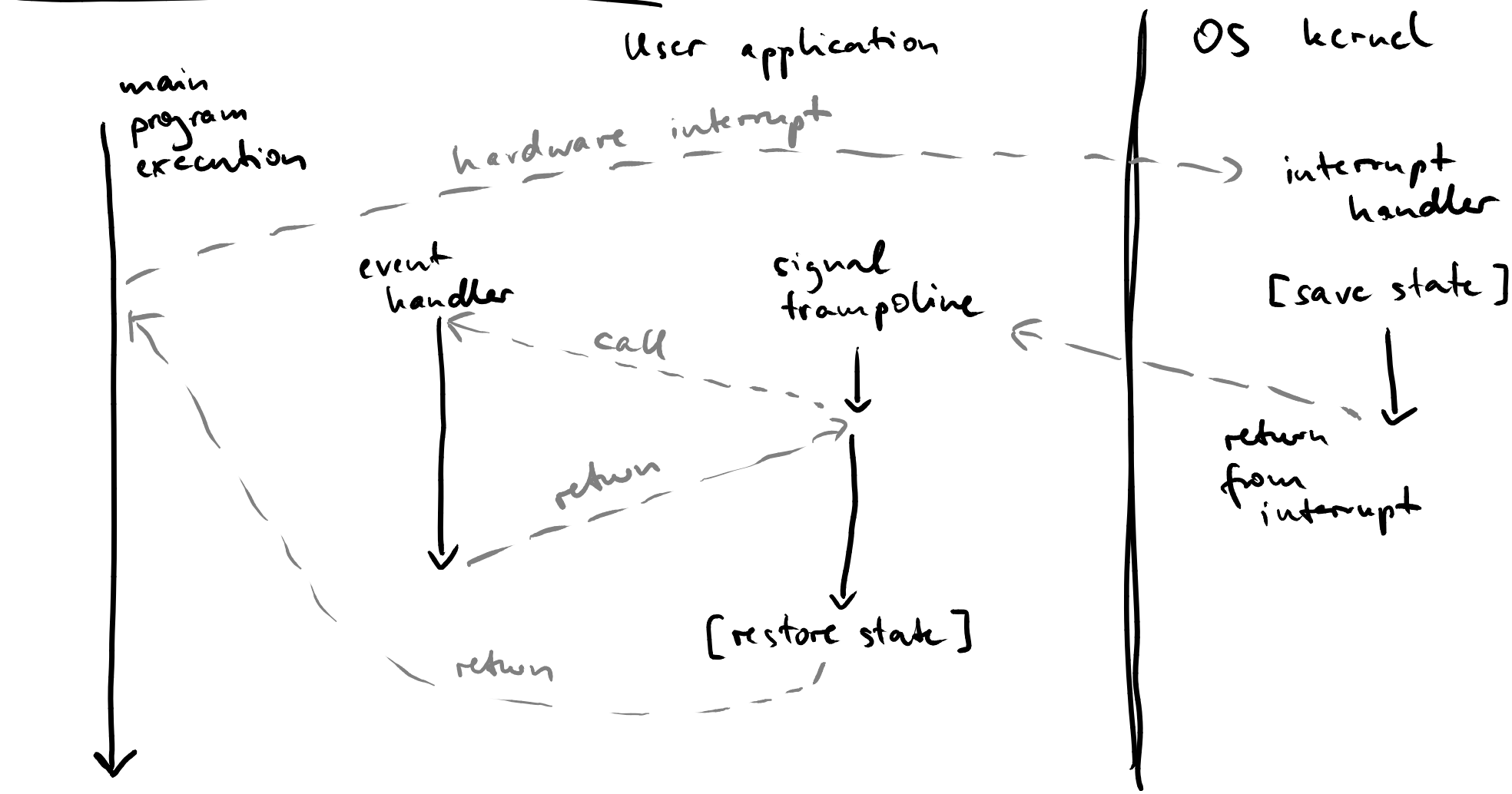
- **List of signals defined by the OS**
- **Use to**
  - Abort a process, e.g., SIGKILL
  - Communicate with a process, e.g., SIGUSR1
- **Program can register a handler to overwrite default behavior**
- **Signals are delivered asynchronously**
  - Current state of program is paused immediately, whereever it is

Example: UNIX Signaling

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

immediately, whereever it is

Example: Signal Delivery



Thread-Based Handlers

- **Specific (background) thread handles events**
- **Often, exactly one thread, to avoid need to synchronize**
- **E.g., GUI thread that reacts to user input and updates UI**
  - Android: UI thread is the “main thread”
  - Only use for short-running operations (otherwise, app becomes unresponsive)

Quiz: Control Abstractions

Which of the following statements is true?

- Coroutines allow for preemptive multi-tasking.
- A calling sequence is the list of subroutines called during an execution.
- Finally-clauses are executed independently of whether an exception is thrown.
- Signals may interrupt the normal execution.

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: Control Abstractions

Which of the following statements is true?

- ~~Coroutines allow for preemptive multi-tasking.~~
- ~~A calling sequence is the list of subroutines called during an execution.~~
- Finally-clauses are executed independently of whether an exception is thrown.
- Signals may interrupt the normal execution.

https://ilias3.uni-stuttgart.de/vote/0ZT9

Overview

- Calling Sequences
- Parameter Passing
- Exception Handling
- Coroutines
- Events ✔