Programming Paradigms

Lecture 13:
Subroutines and Control Abstraction (Part 2)
Wake-up Exercise

What does the following Java code print?

```java
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?

```java
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?

```java
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`
Wake-up Exercise

What does the following Java code print?

```java
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.
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.");
}
Wake-up Exercise

What does the following Java code print?

```java
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.");
}
```

Not a NullPointerException anymore: Nothing caught here

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

What does the following Java code print?

```java
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
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++):

```cpp
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++):

```cpp
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

```java
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

```java
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

```java
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

```cpp
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

```cpp
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:
    ```plaintext
    EXCEPTION empty_queue
    ```

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

```python
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?

```python
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
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 transferring to another coroutines
- When transferring 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 transferring to another coroutines
- **When transferring 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 coroutine runs at a time

- Control flow transferred 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
L
n
B
D
s
Q
C
R
L

→ static links

S
B
D
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, wherever it is
Example: UNIX Signaling

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

- Main program execution
- User application
  - Hardware interrupt
  - Event handler
  - Call
  - Signal trampoline
  - Return
  - [Restore state]

- OS kernel
  - Interrupt handler
  - [Save state]
  - Return from interrupt
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.
Overview

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

✓