CS49K Programming Languages

Chapter 9: Subroutines and Control Abstraction

LECTURE 12: OTHER SUBROUTINES

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Exception
- Assertion
- Unit Test
- Coroutine
- Event Callbacks – Signal and Slots
- Lambda Function
- Runnable

Exception Handling I

What is an exception?

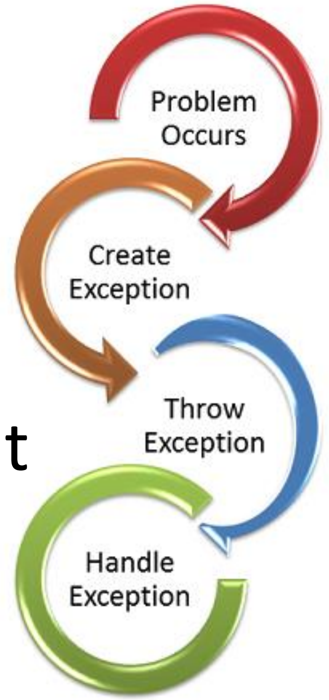
SECTION 1

Exception Handling

- What is an exception?
  - a hardware-detected run-time error or unusual condition detected by software
- Examples
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions, not necessarily errors

Exception Handling

- What is an exception handler?
  - code executed when exception occurs
  - may need a different handler for each type of exception
- Why design in exception handling facilities?
  - allow user to explicitly handle errors in a uniform manner
  - allow user to handle errors without having to check these conditions explicitly in the program everywhere they might occur
- Exception Handling is a special subroutine call.



Work-Arounds

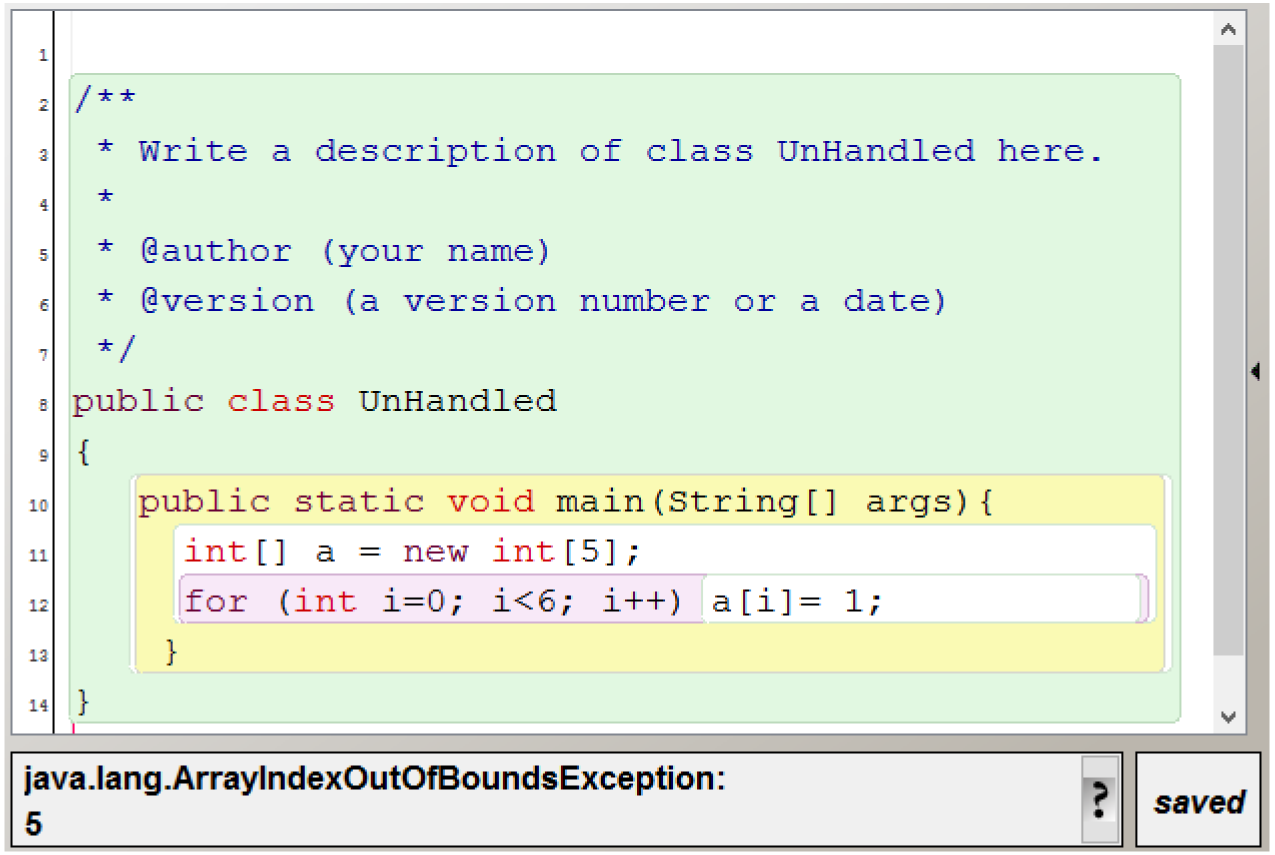
To cope with such errors without an exception-handling mechanism, the programmer has basically three options, none of which is entirely satisfactory:

1. “Invent” a value that can be used by the caller when a real value could not be returned.

2. Return an explicit “status” value to the caller, who must inspect it after every call. The status may be written into an extra, explicit parameter, stored in a global variable, or encoded as otherwise invalid bit patterns of a function’s regular return value.

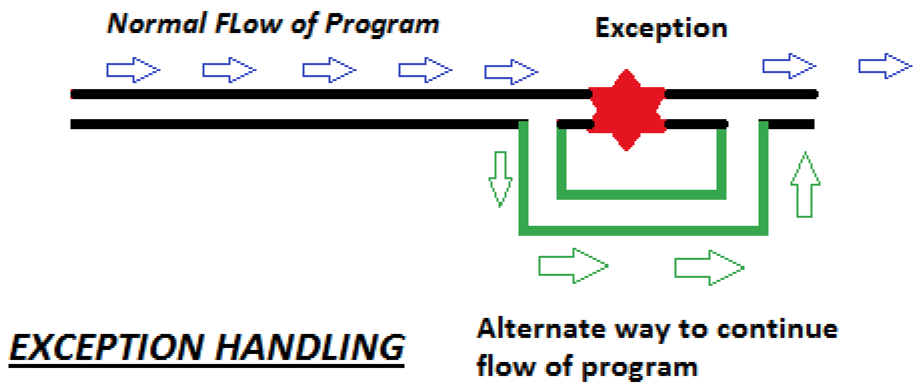
3. Rely on the caller to pass a closure (in languages that support them) for an error-handling routine that the normal routine can call when it runs into trouble.

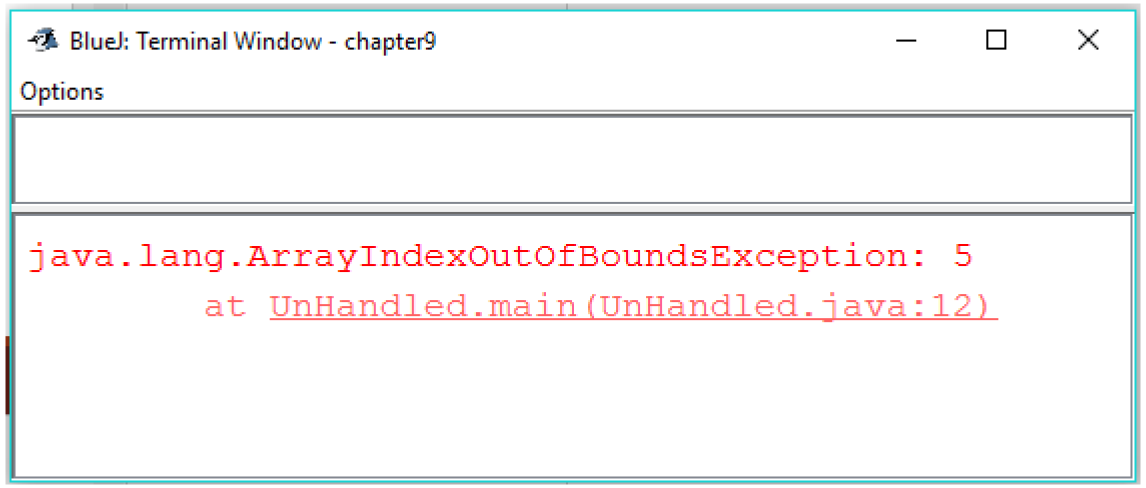
Unhandled Exception



```java
/**
 * Write a description of class UnHandled here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class UnHandled
{
    public static void main(String[] args){
      int[] a = new int[5];
      for (int i=0; i<6; i++) a[i]= 1;
    }
}
```

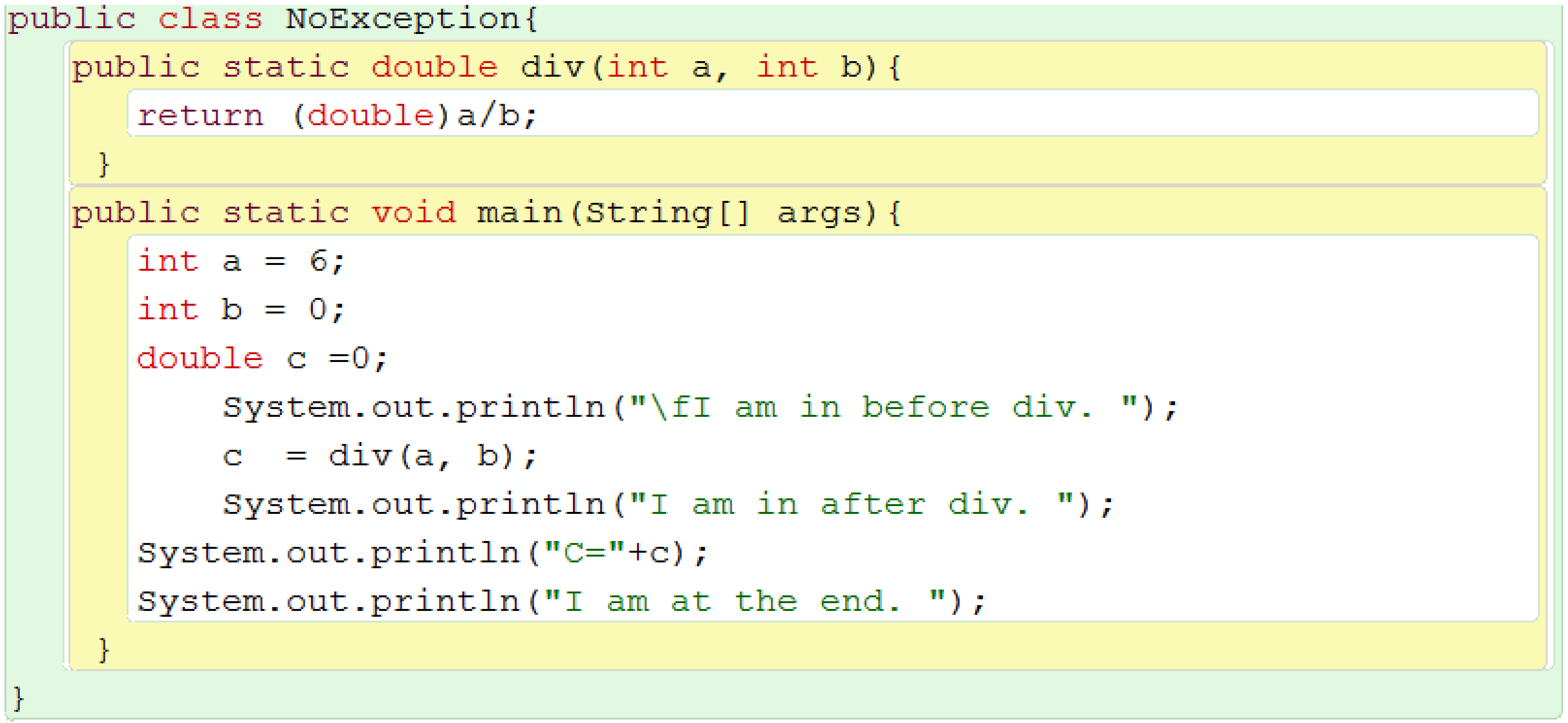
```
java.lang.ArrayIndexOutOfBoundsException:
5
```



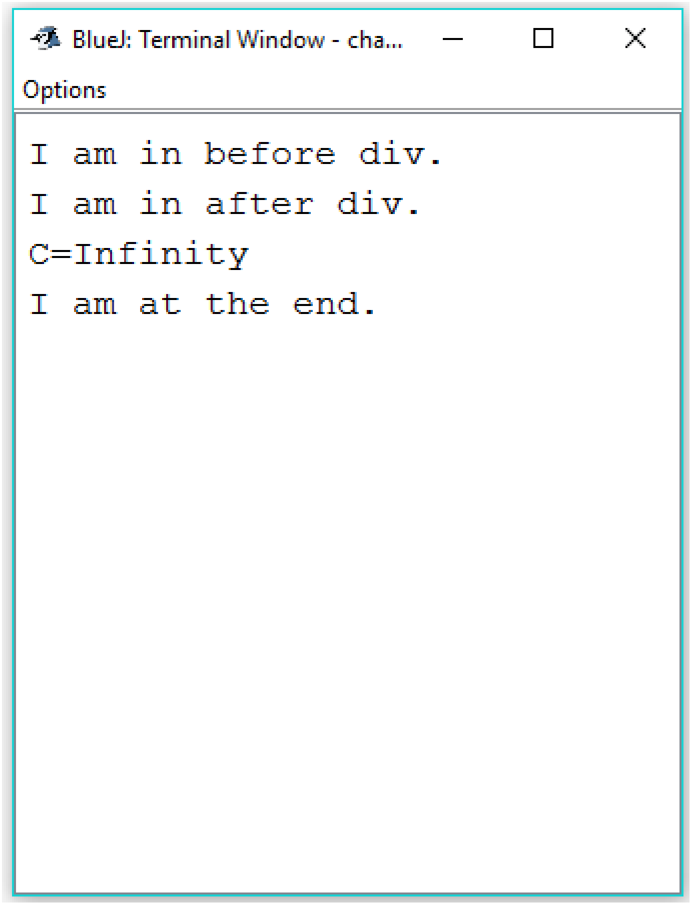


```
java.lang.ArrayIndexOutOfBoundsException: 5
        at UnHandled.main(UnHandled.java:12)
```

Unchecked Overflow

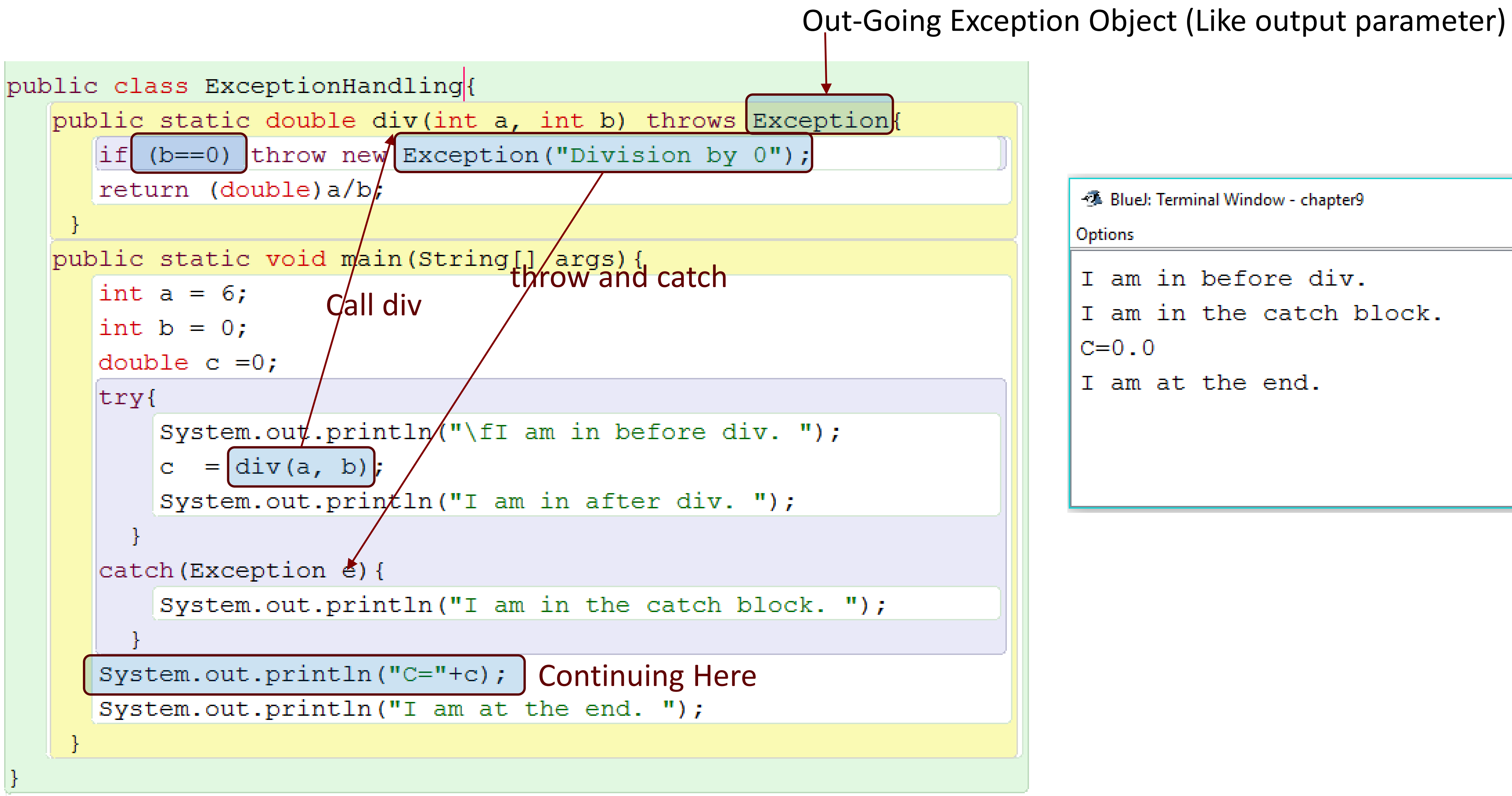


```
public class NoException{
    public static double div(int a, int b){
        return (double)a/b;
    }
    public static void main(String[] args){
        int a = 6;
        int b = 0;
        double c =0;
            System.out.println("\fI am in before div. ");
            c  = div(a, b);
            System.out.println("I am in after div. ");
        System.out.println("C="+c);
        System.out.println("I am at the end. ");
    }
}
```

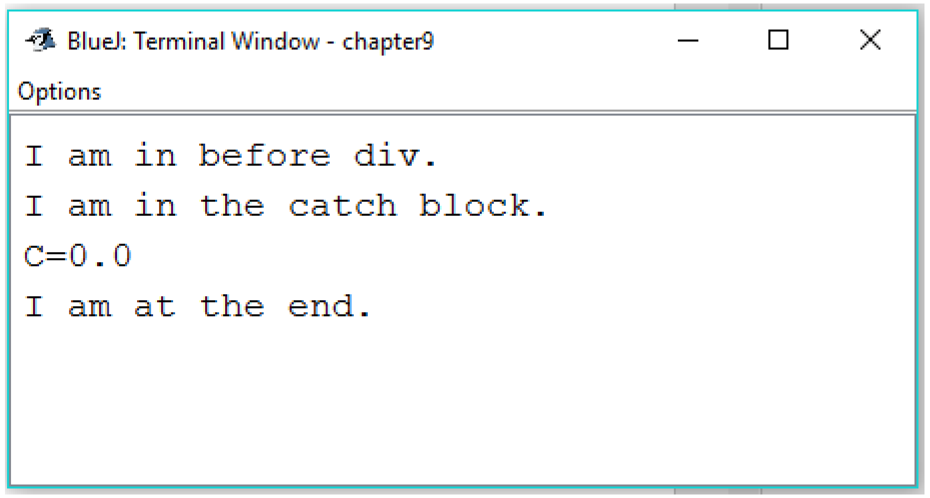


```
I am in before div.
I am in after div.
C=Infinity
I am at the end.
```

Handled User-Defined Exception

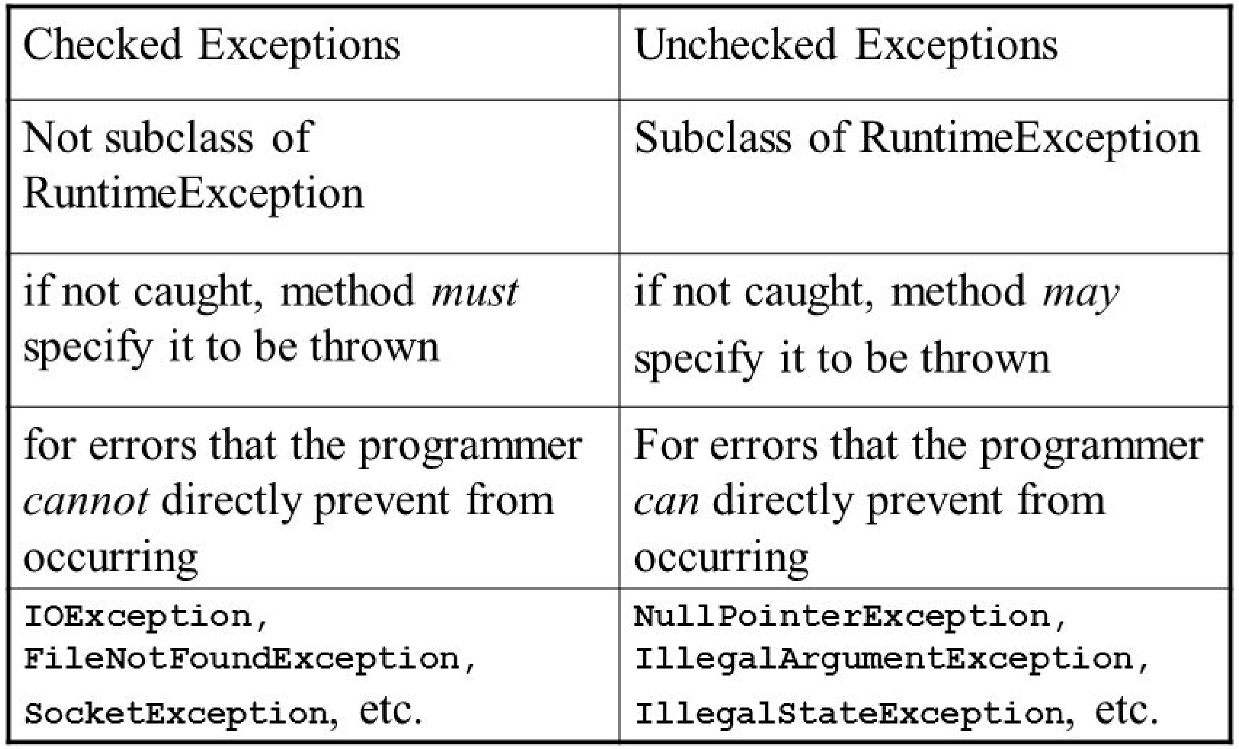


```
public class ExceptionHandling{
    public static double div(int a, int b) throws Exception{
        if (b==0) throw new Exception("Division by 0");
        return (double)a/b;
    }
    public static void main(String[] args){
        int a = 6;
        int b = 0;
        double c =0;
        try{
            System.out.println("\fI am in before div. ");
            c  = div(a, b);
            System.out.println("I am in after div. ");
        }
        catch(Exception e){
            System.out.println("I am in the catch block. ");
        }
        System.out.println("C="+c);
        System.out.println("I am at the end. ");
    }
}
```



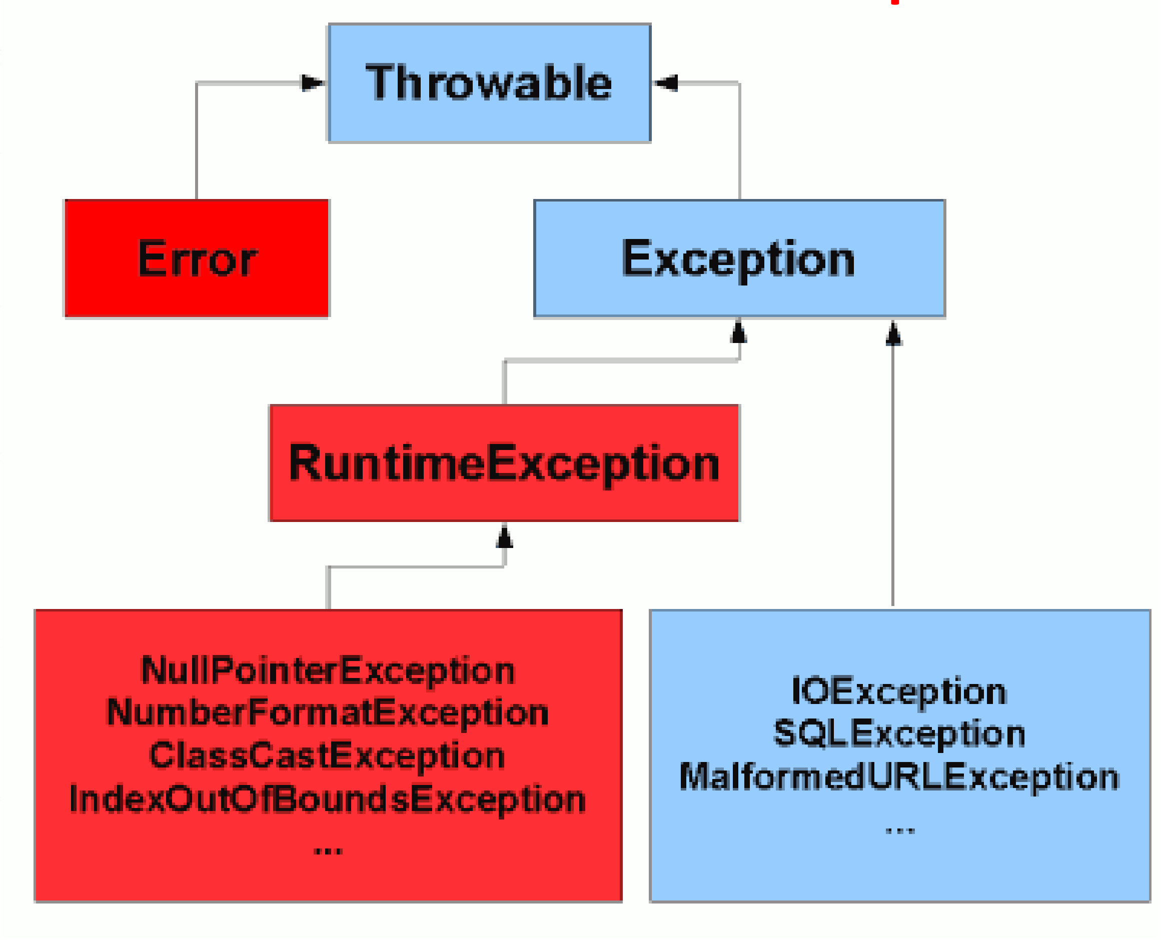
```
I am in before div.
I am in the catch block.
C=0.0
I am at the end.
```

Checked vs. Unchecked Exceptions



Checked Exceptions	Unchecked Exceptions
Not subclass of RuntimeException	Subclass of RuntimeException
if not caught, method *must* specify it to be thrown	if not caught, method *may* specify it to be thrown
for errors that the programmer *cannot* directly prevent from occurring	For errors that the programmer *can* directly prevent from occurring
`IOException`, `FileNotFoundException`, `SocketException`, etc.	`NullPointerException`, `IllegalArgumentException`, `IllegalStateException`, etc.

Checked Versus Unchecked Exceptions



Exception Handling II

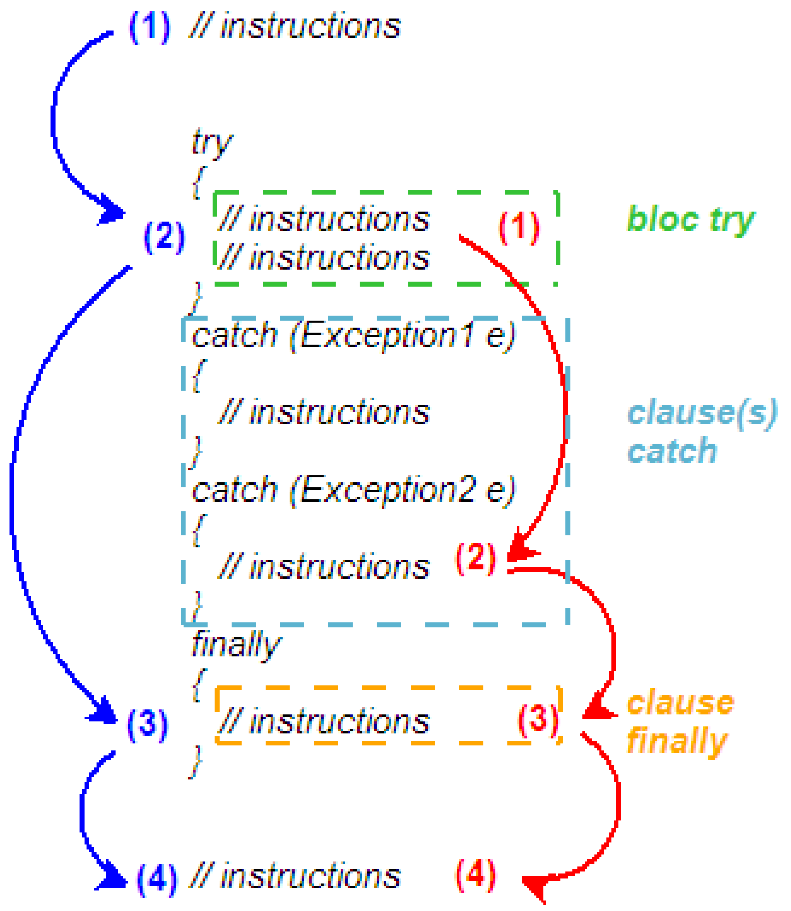
Try-Catch-Finally blocks

SECTION 2

C++ Try Block

try-catch-finally

```
try {
    ...
    if (something_unexpected)
        throw my_exception();
    ...
    cout << "everything's ok\n";
    ...
} catch (my_exception) {
    cout << "oops\n";
}
```



Propagation of an Exception out of a Called Routine (C++)

```
try {
    ...
    foo();
    ...
    cout << "everything's ok\n";
    ...
} catch (my_exception) {
    cout << "oops\n";
}
```

```
void foo() {
    ...
    if (something_unexpected)
        throw my_exception();
    ...
}
```

Defining Exceptions

Ada:

```
declare empty_queue : exception;
```

Modula-3:

```
EXCEPTION empty_queue;
```

Declaration of Exception:

```
class empty_queue {};
```

Exception with Parameter:

```
class duplicate_in_set {                        // C++
    item dup;           // element that was inserted twice
};
...
throw duplicate_in_set(d);
```

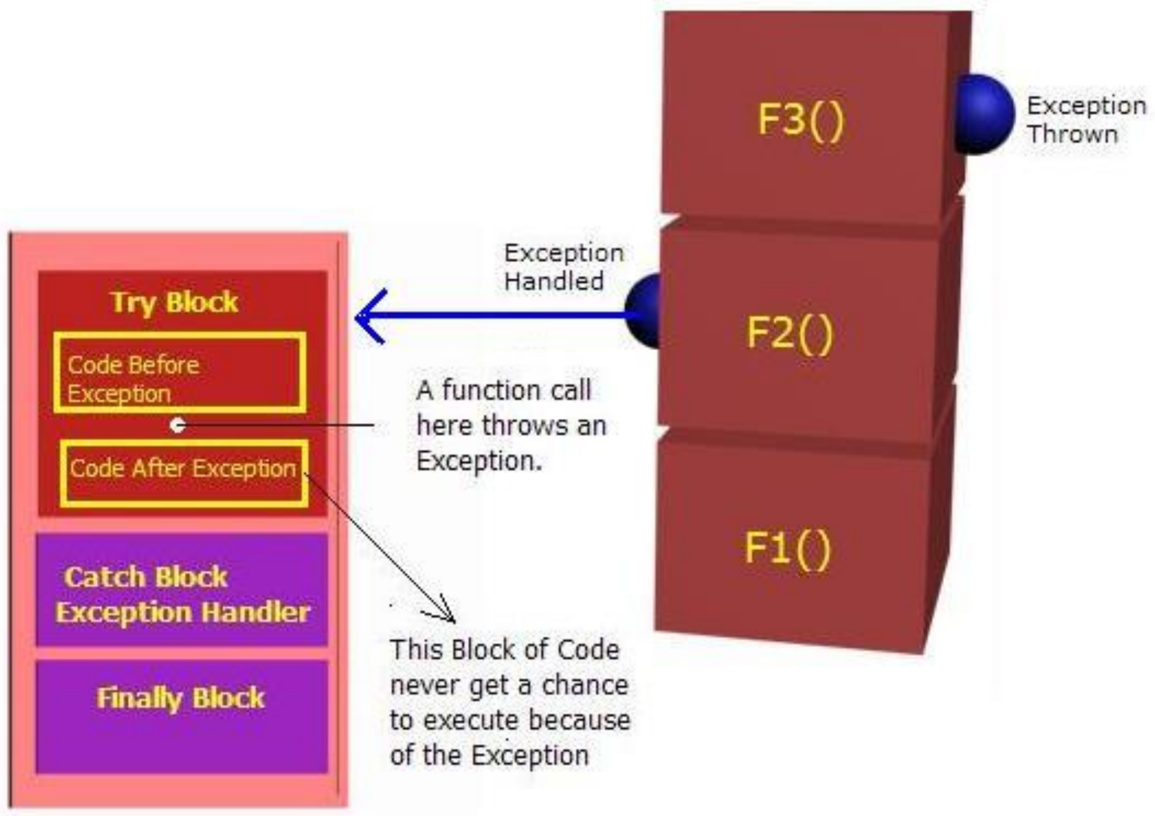
Exception Propagation

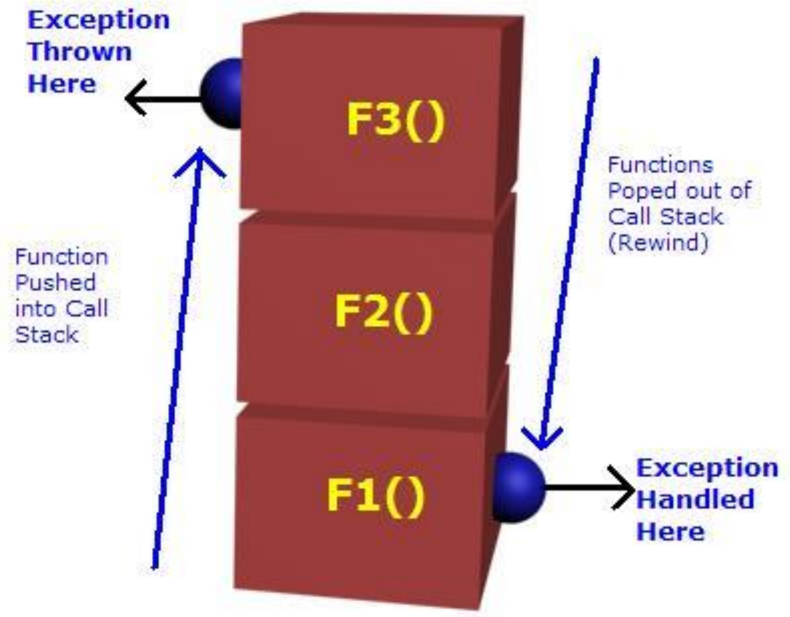
```
try {                     // try to read from file
    ...
    // potentially complicated sequence of operations
    // involving many calls to stream I/O routines
    ...
} catch(end_of_file) {
    ...
} catch(io_error e) {
    // handler for any io_error other than end_of_file
    ...
} catch(...) {
    // handler for any exception not previously named
    // (in this case, the triple-dot ellipsis is a valid C++ token;
    // it does not indicate missing code)
}
```

Exception handler in ML:

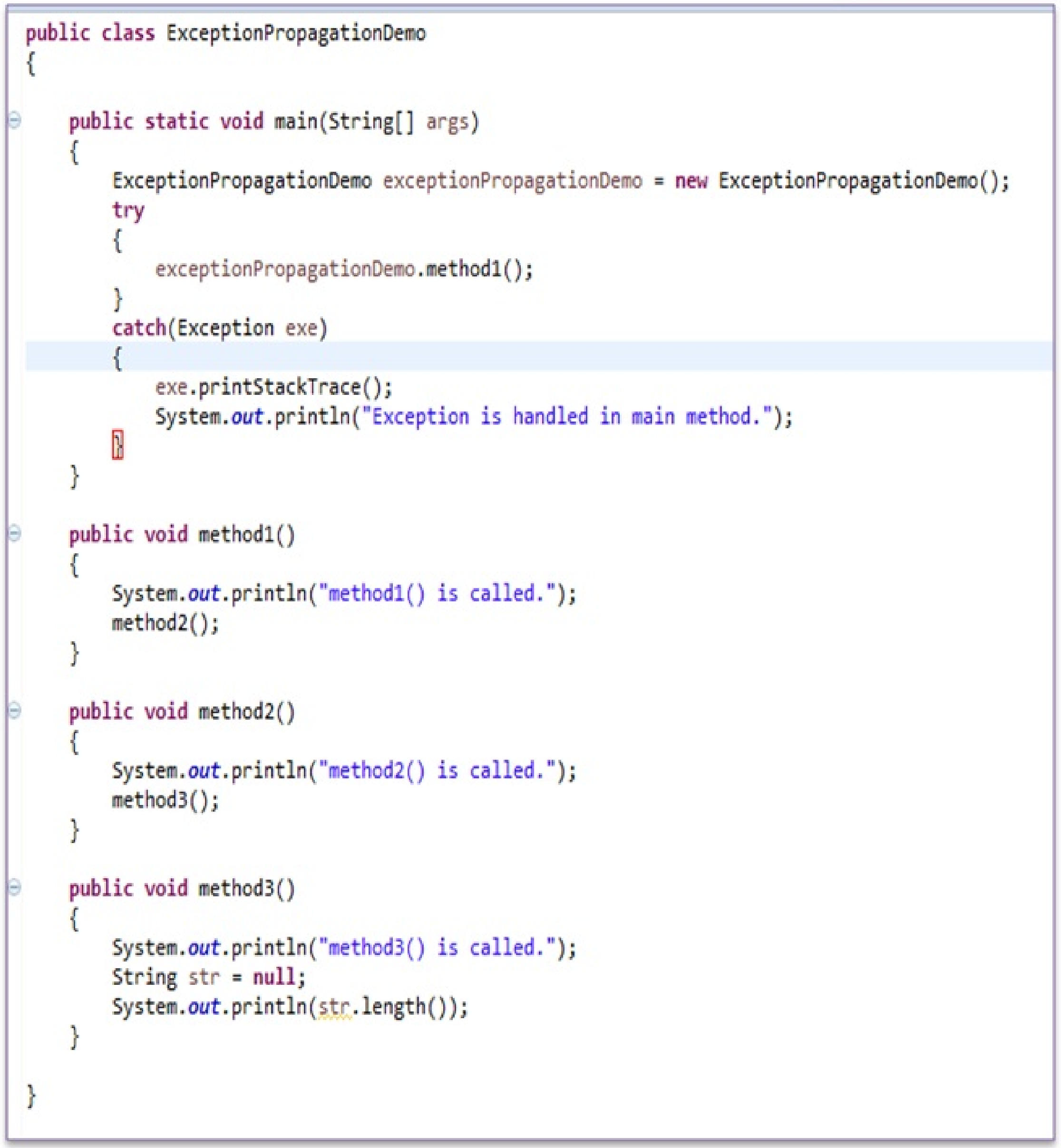
```
val foo = (f(a) * b) handle Overflow => max_int;
```

Handlers on Expressions

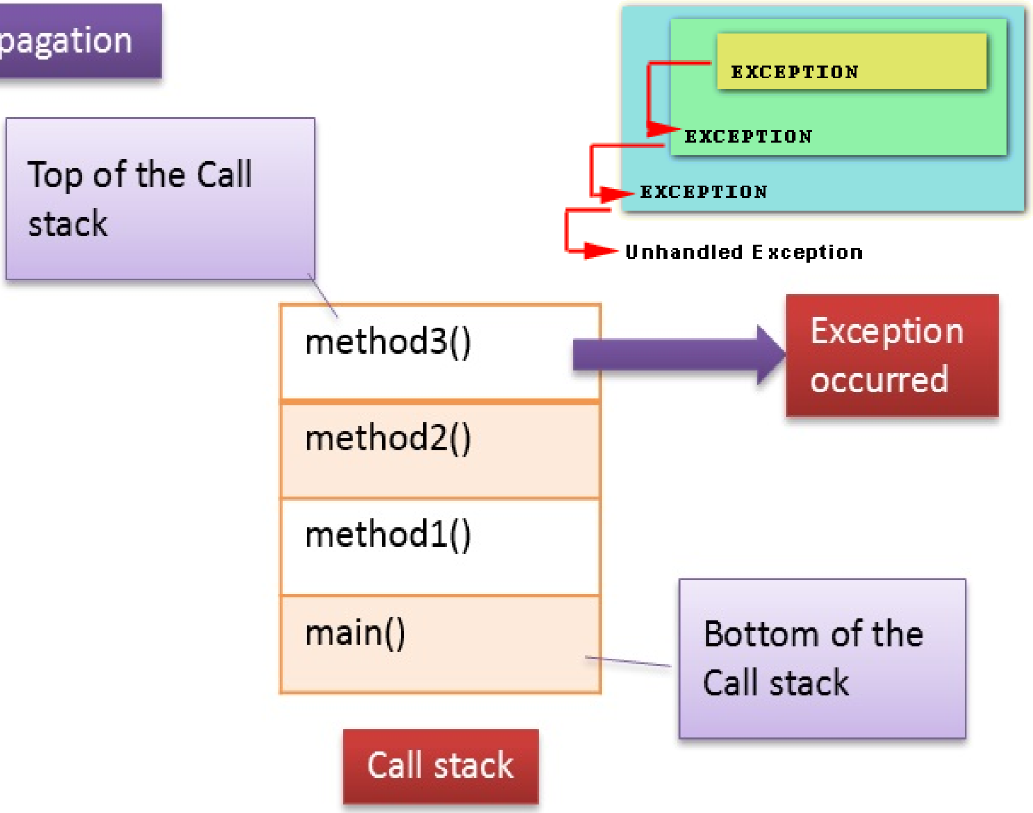

Try Block
Code Before Exception
Code After Exception
Catch Block Exception Handler
Finally Block
Exception Handled
A function call here throws an Exception.
This Block of Code never get a chance to execute because of the Exception
F3()
Exception Thrown
F2()
F1()


Exception Thrown Here
Function Pushed into Call Stack
F3()
F2()
F1()
Functions Poped out of Call Stack (Rewind)
Exception Handled Here

Java Exception propagation



```
public class ExceptionPropagationDemo
{

    public static void main(String[] args)
    {
        ExceptionPropagationDemo exceptionPropagationDemo = new ExceptionPropagationDemo();
        try
        {
            exceptionPropagationDemo.method1();
        }
        catch(Exception exe)
        {
            exe.printStackTrace();
            System.out.println("Exception is handled in main method.");
        }
    }

    public void method1()
    {
        System.out.println("method1() is called.");
        method2();
    }

    public void method2()
    {
        System.out.println("method2() is called.");
        method3();
    }

    public void method3()
    {
        System.out.println("method3() is called.");
        String str = null;
        System.out.println(str.length());
    }

}
```



- ✓ In this example exception occurs in method3() where it is not handled, so it is propagated to method2 where it is not handled, again it is propagated to method1() where exception is not handled, again it is propagated to main() method where exception is handled.
- ✓ Exception can be handled in any method in call stack either in method3(), method2() method1() or main() method.

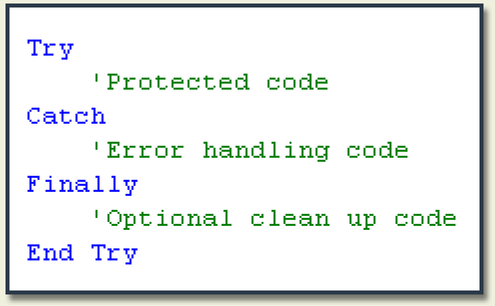
Clean Up Operations

Finally Code Block

Python finally Clause:

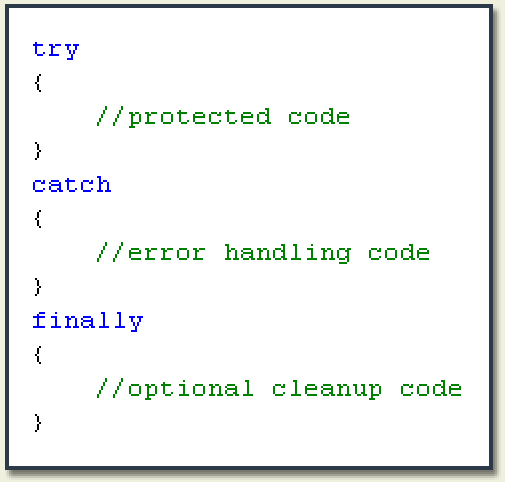
```
try:                                          # protected block
    my_stream = open(“foo.txt”, “r”);         # “r” means for reading
    for line in my_stream:
    …
finally:
     my_stream.close()
```

VB.NET:



```
Try
    'Protected code
Catch
    'Error handling code
Finally
    'Optional clean up code
End Try
```

C#:



```
try
{
    //protected code
}
catch
{
    //error handling code
}
finally
{
    //optional cleanup code
}
```

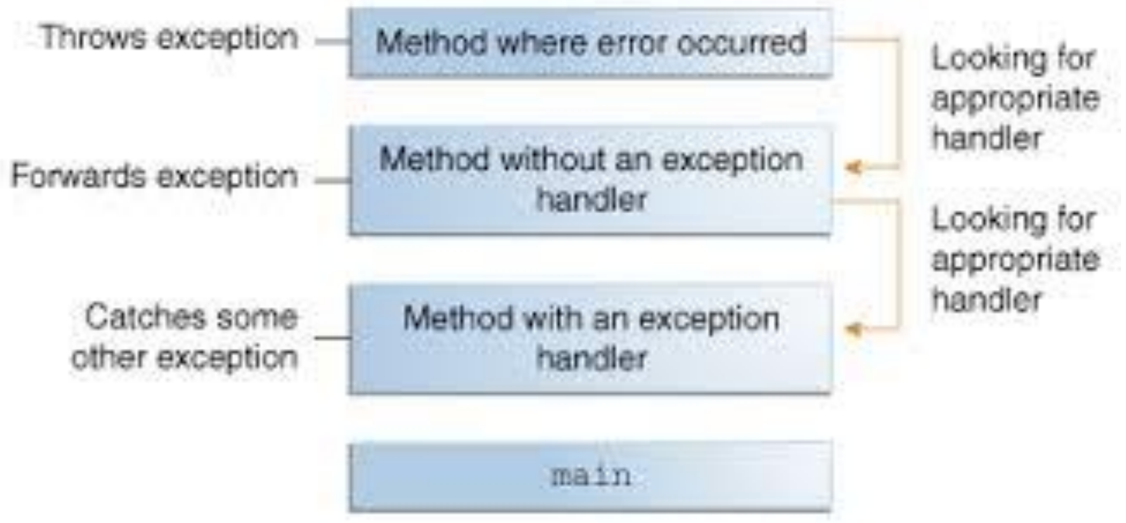
Implementation of Exceptions

Stacked Exception Handlers:

```
if exception matches duplicate_in_set
    . . .
else
    reraise exception
```

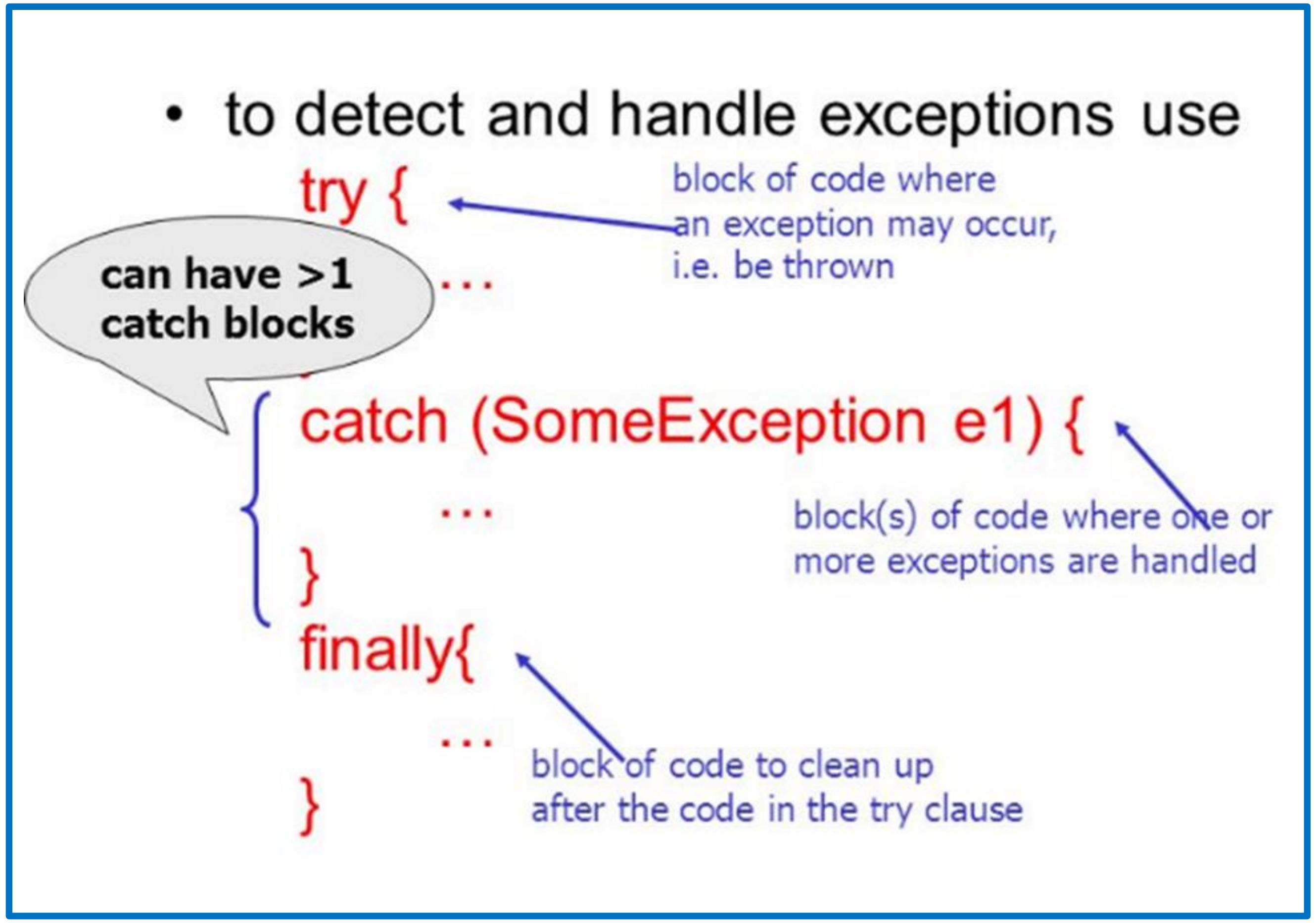
Multiple Exception Per Handler:

```
if exception matches end_of_file
    . . .
elsif exception matches io_error
    . . .
else
    . . .          -- "catch-all" handler
```


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

1. Locate the Handler
2. Pass the exception object to catch blocks
   (catch block is a set of if-else if- else clocks or switch)

Implementation of Exceptions



- to detect and handle exceptions use

```
try {
    ...
}
catch (SomeException e1) {
    ...
}
finally{
    ...
}
```

block of code where an exception may occur, i.e. be thrown
can have >1 catch blocks
block(s) of code where one or more exceptions are handled
block of code to clean up after the code in the try clause

```
if (exception_condiftion)
      throw new SomeException();
```

```
if (exception_condiftion) {
      locate catch_function();
      catch_function(new SomeException());
}
```

```
void catch_function(SomeException e1){
   ...
}
```

Coroutines

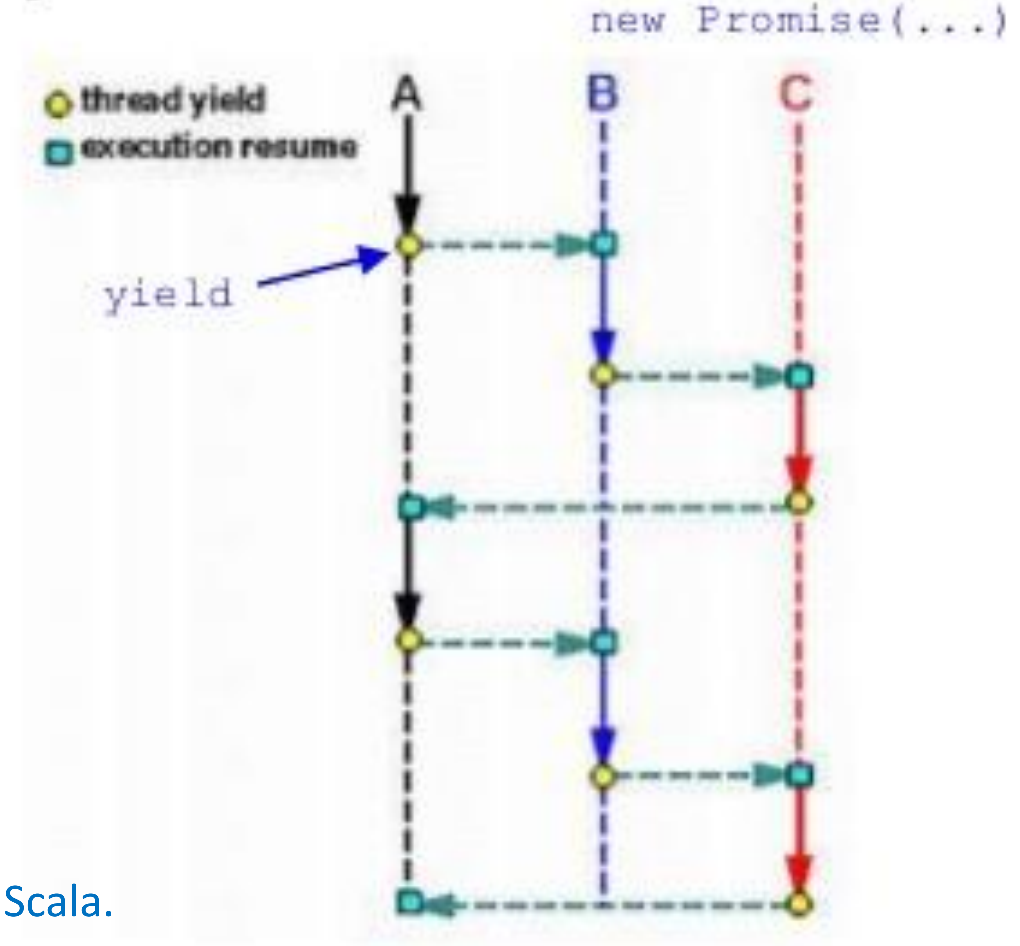
SECTION 3

Coroutines

- Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name
- Coroutines can be used to implement
  - iterators (Section 6.5.3)
  - threads (to be discussed in Chapter 13)
- Because they are concurrent (i.e., simultaneously started but not completed), coroutines **cannot** share a single stack

Coroutines = “Cooperative Routines”

- Generators provide *cooperation* through the *yield* keyword
  - Explicitly tells IO-Loop (uv, etc.) to suspend execution
- Promises provide the *routines*
  - Scheduled and run asynchronously by IO-Loop
- Generators + Promises = Cooperation + Routines = Coroutines!



Ada, Java, C#, C++, Python, Ruby, Haskell, Go, and Scala.

https://www.cs.mtu.edu/~shene/NSF-3/e-Book/FUNDAMENTALS/thread-yield.jpg

Coroutines

- Like a **continuation**, a **coroutine** is represented by a closure (a code address and a referencing environment), into which we can jump by means of a non-local **`goto`**, in this case a special operation known as **`transfer`**.
- The principal difference between the two abstractions is that a **continuation** is a constant—it does not change once created—while a **coroutine** changes every time it runs.
- When we transfer from one coroutine to another, our old program counter is saved: the coroutine we are leaving is updated to reflect it.
- Thus, if we perform a **`goto`** into the same continuation multiple times, each jump will start at precisely the same location, but if we perform a transfer into the same coroutine multiple times, each jump will take up where the previous one left off.

Interleaving Coroutines

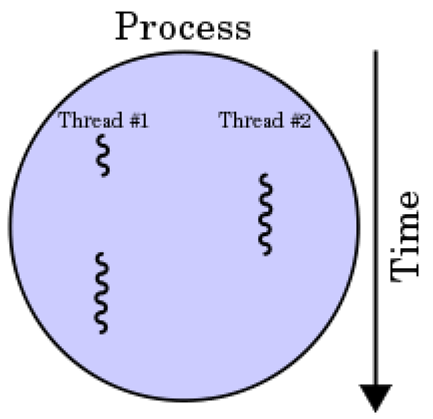
```
us, cfs : coroutine

coroutine check_file_system
    -- initialize
    detach
    for all files
        . . .
         transfer(us)
         . . .
              transfer(us)
         . . .
         transfer(us)
         . . .
```

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

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

Threads and Coroutines

- It is easy to build a simple thread package given coroutines.
- Most programmers would agree, however, that threads are substantially easier to use, because they eliminate the need for explicit transfer operations.
- This contrast—a lot of extra functionality for a little extra implementation complexity—probably explains why coroutines as an explicit programming abstraction are relatively rare.

A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
Thread is conceptually concurrent. (Parallel or not parallel.)



Stack Allocation

- Because they are concurrent, coroutines cannot share a single stack: their subroutine calls and returns, taken as a whole, do not occur in last-in-first-out order.
- If each coroutine is declared at the outermost level of lexical nesting, then their stacks are entirely disjoint: the only objects they share are global, and thus statically allocated.
- Most operating systems make it easy to allocate one stack, and to increase its portion of the virtual address space as necessary during execution. It is usually not easy to allocate an arbitrary number of such stacks; space for coroutines is something of an implementation challenge.

Stack Allocation

Implementation 1: Give each coroutine a fixed amount of statically allocated stack space. (Overflow? Waste?)

Implementation 2: Put in heap. If stack frames are allocated from the heap, as they are in most functional languages, then the problems of overflow and internal fragmentation are avoided. At the same time, the overhead of each subroutine call is significantly increased.

Implementation 3: Mixed. Allocate the stacks in chunks.

Cactus Stacks

1. Use Dynamic Links and static links.
2. Static Part may co-exist.
3. Dynamic Part is Last In First Out.

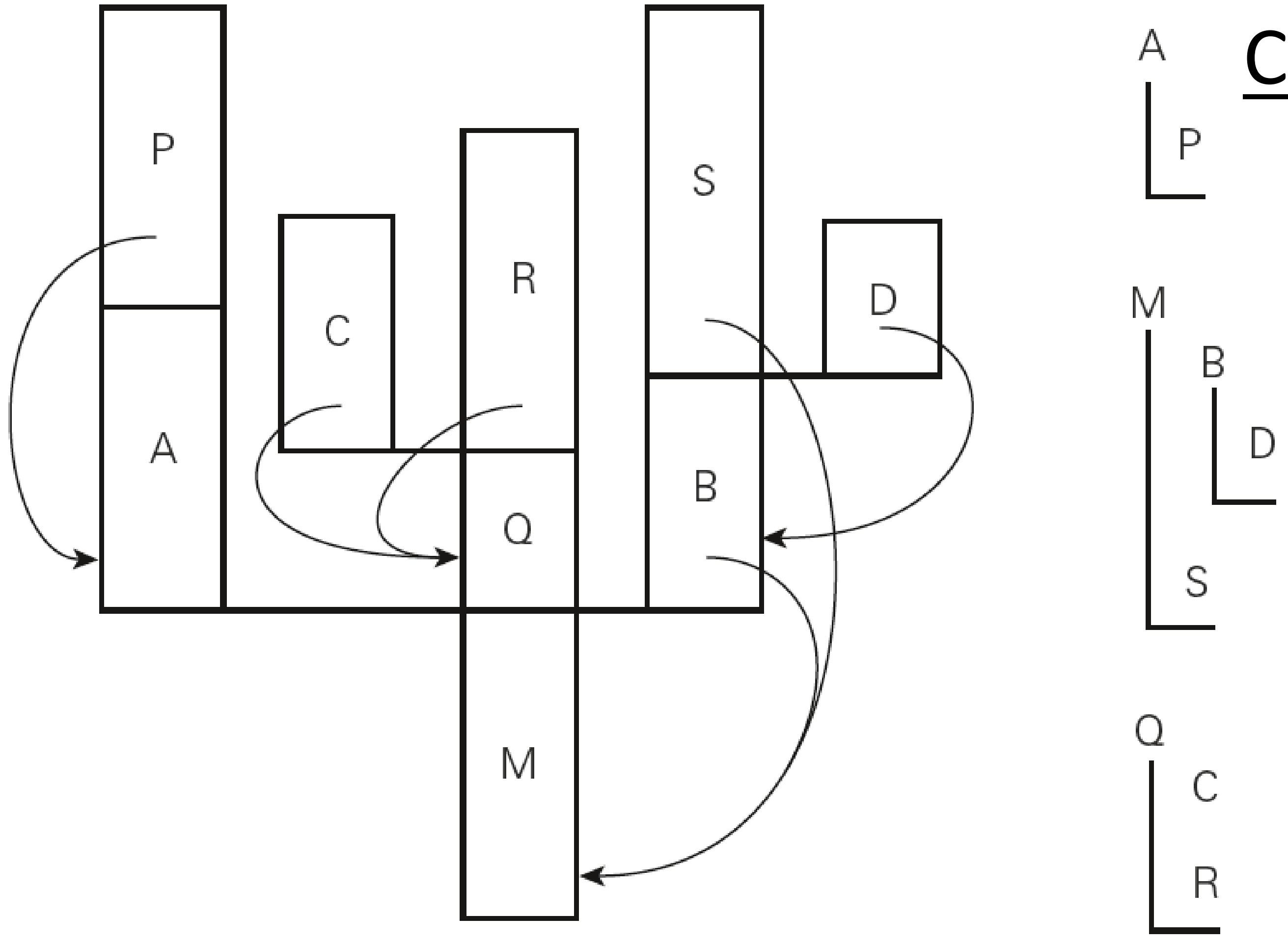
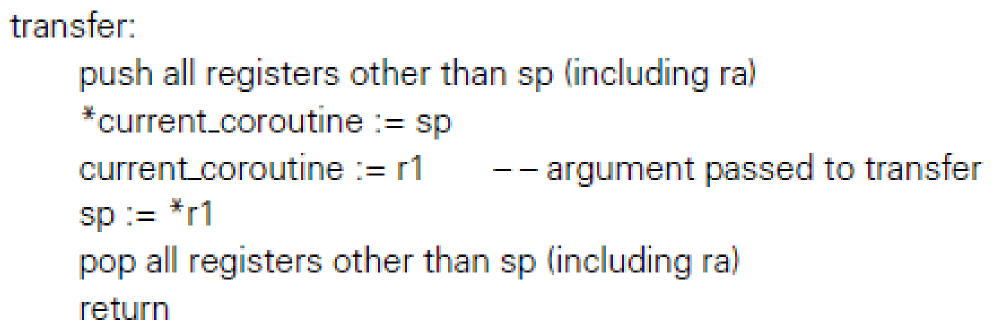


Figure 9.4 **A cactus stack.** Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it. (Coroutine B, for example, was created by the main program, M. B in turn called subroutine S and created coroutine D.)

Transfer

- To transfer from one coroutine to another, the run-time system must change the program counter (PC), the stack, and the contents of the processor's registers.
- These changes are encapsulated in the **`transfer`** operation: one coroutine calls transfer; a different one returns.
- **Change Stack among Coroutines:** simply to change the stack pointer register, and to avoid using the frame pointer inside of transfer itself. At the beginning of transfer we push the return address and all of the other callee saves registers onto the current stack. We then change the **`sp`**, pop the (**`new`**) return address (**`ra`**) and other registers off the new stack, and return:



```
transfer:
    push all registers other than sp (including ra)
    *current_coroutine := sp
    current_coroutine := r1        -- argument passed to transfer
    sp := *r1
    pop all registers other than sp (including ra)
    return
```

Events

SECTION 4

What is an Event?

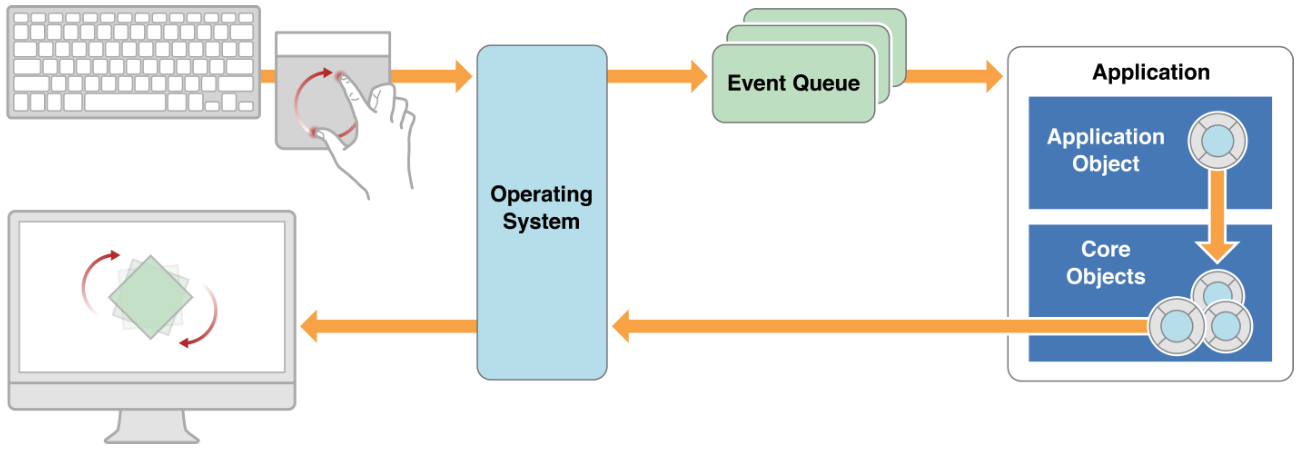
An **event** is something to which a running program (a process) needs to respond but which occurs outside the program, at an unpredictable time.

The most common events are inputs to a graphical user interface (GUI) system: keystrokes mouse motions, button clicks. They may also be network operations or other asynchronous I/O activity: the arrival of a message, the completion of a previously requested disk operation.

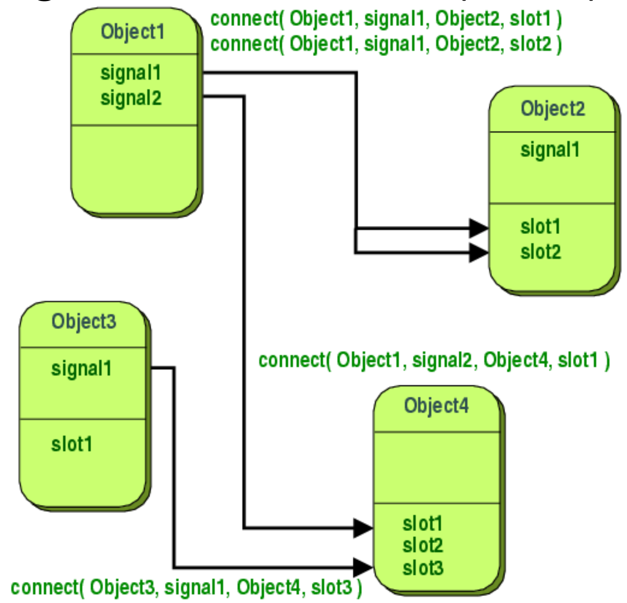
In the I/O operations, we assumed that a program looking for input will request it explicitly, and will wait if it isn't yet available. This sort of synchronous (at a specified time) and blocking (potentially wait-inducing) input is generally not acceptable for modern applications with graphical interfaces. Instead, the programmer usually wants a handler—a special subroutine—to be invoked when a given event occurs.

Handlers are sometimes known as **callback** functions, because the run-time system calls back into the main program instead of being called from it. In an object-oriented language, the callback function may be a method of some handler object, rather than a static subroutine.

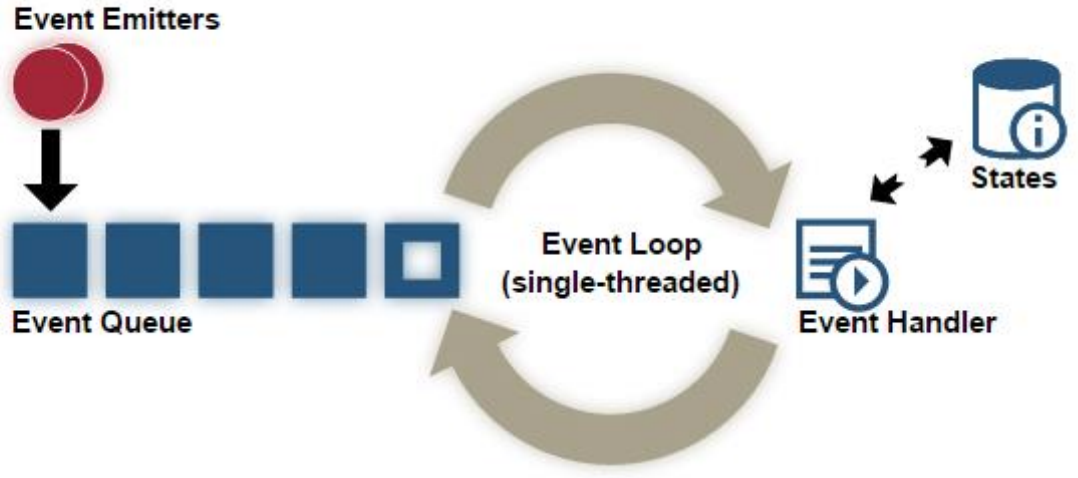
Event-Driven Programming Environment



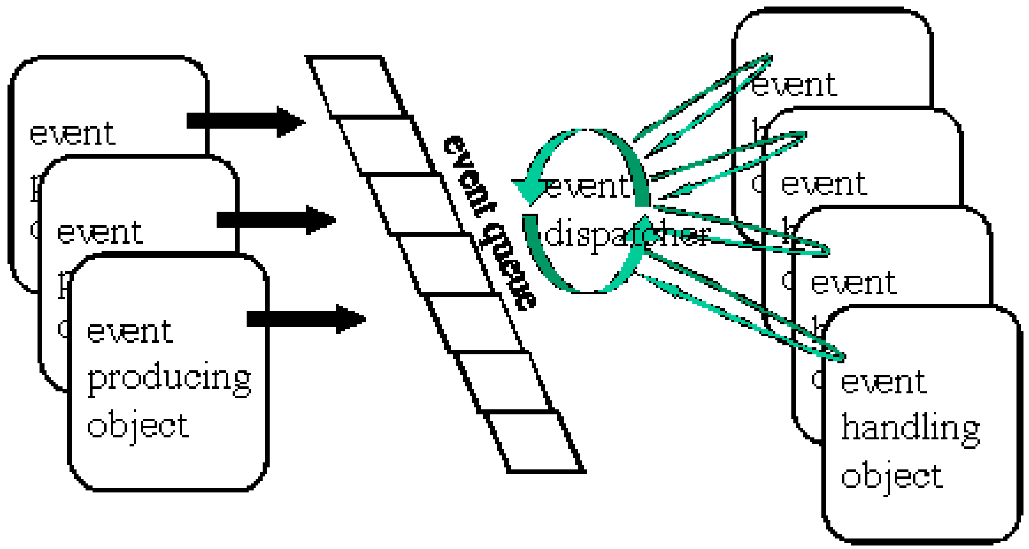
Signal-Slot Mechanism (C/C++)

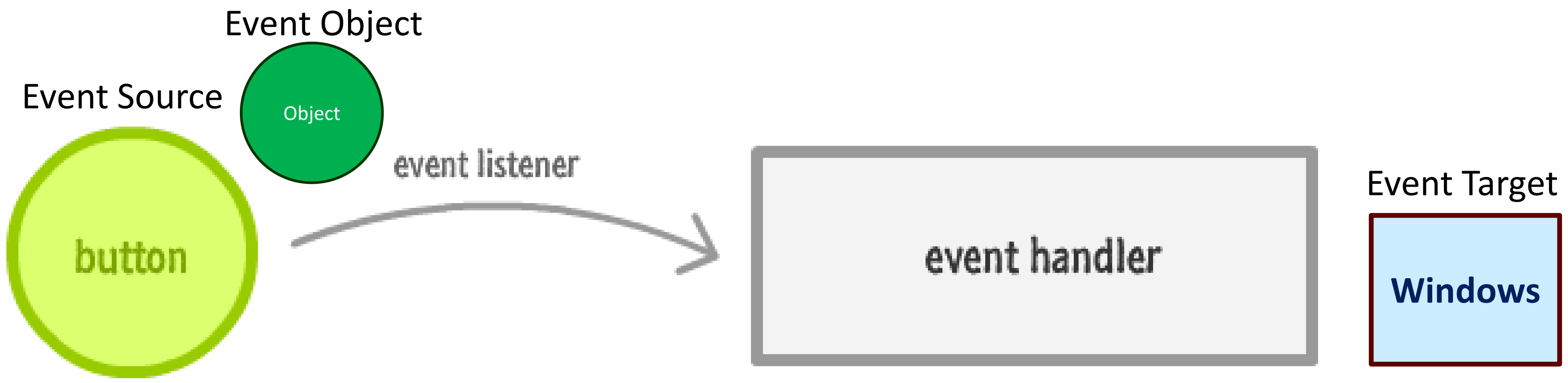

Object1
signal1
signal2
connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Single-Threaded Event Loop (Javascript)



Multi-Threaded Event Dispatcher (Java)



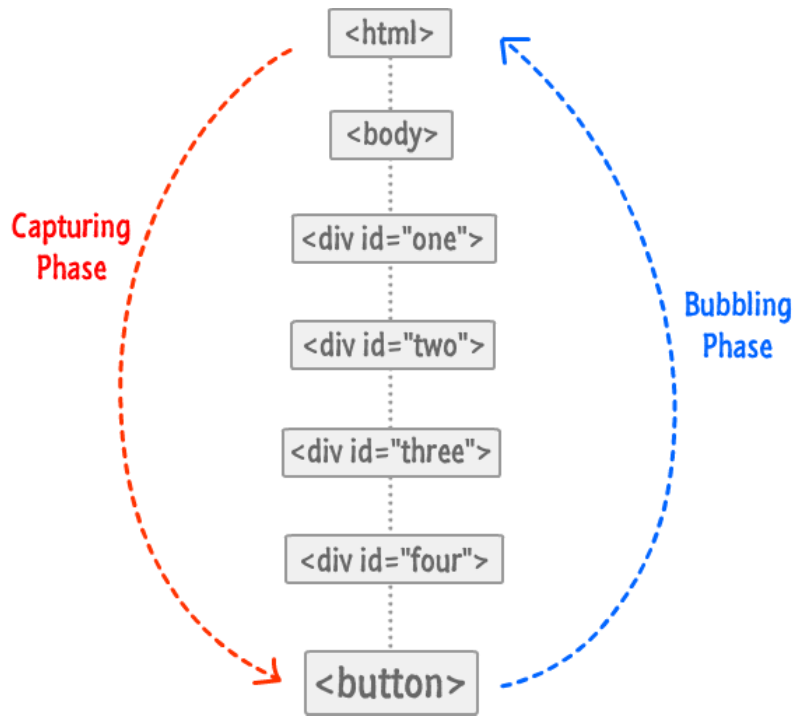


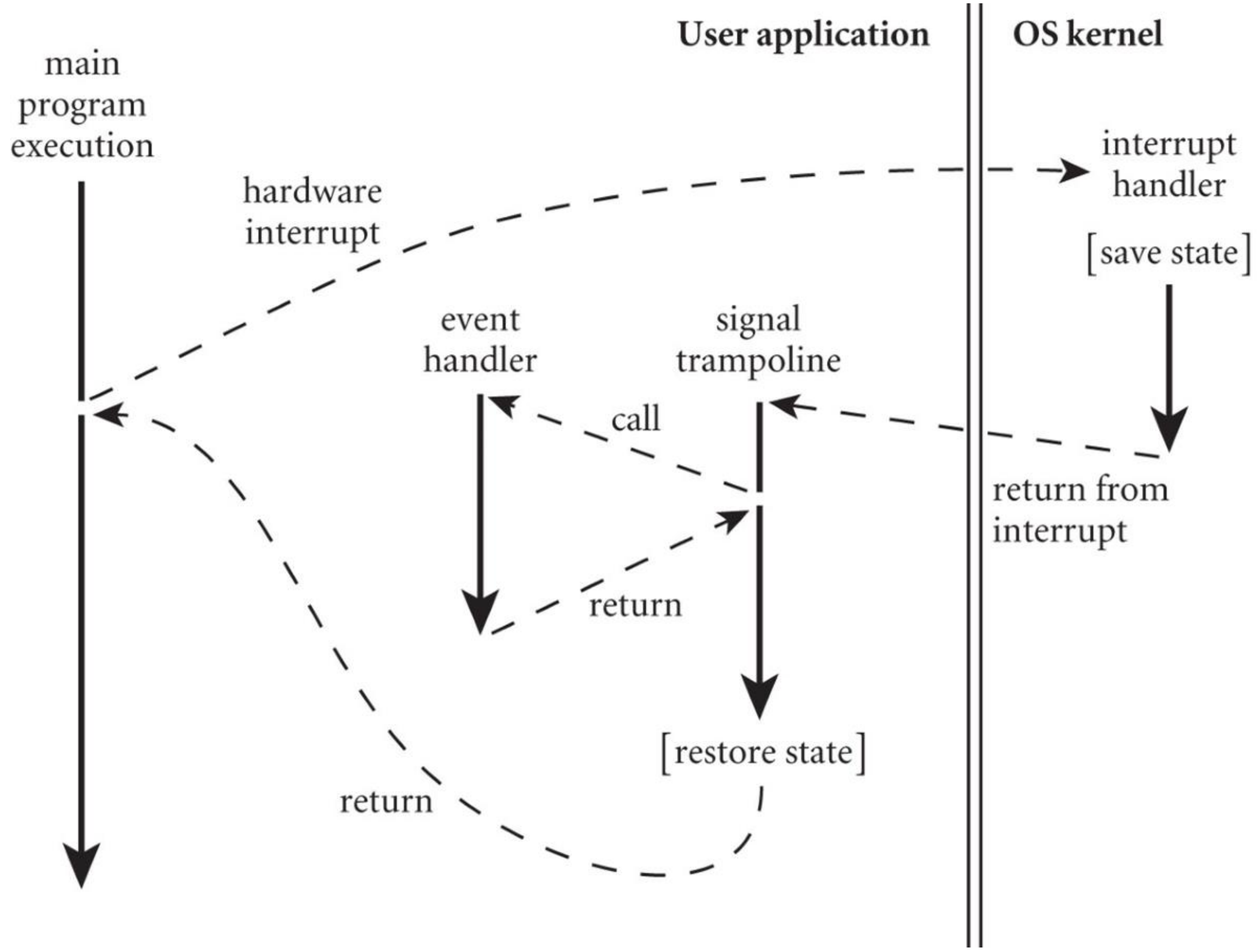
Ways of Event Connections:

1. Default Event (Mouse Click)
2. Connected Events (Moue Button Down, Mouse Button Release, Mouse Drag, …)
3. Custom-Designed Events

Handler Type:

1. actionPerformed (Default)
2. mousePressed, mouseReleased (by event type)
3. Custom-defined handler
4. Slot function
5. Lambda expression
6. Call-back function




User application
OS kernel
main program execution
hardware interrupt
interrupt handler
[save state]
event handler
signal trampoline
call
return from interrupt
return
[restore state]
return

Sequential Handlers

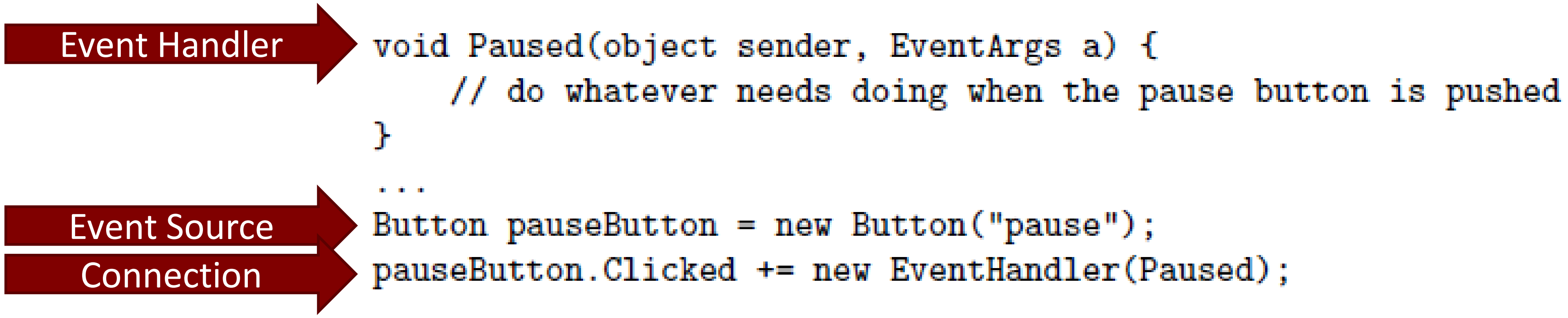
- setup_handler (P): a program invokes a library routine, passing as argument the subroutine it want to have invoked when the event occurs.
- In practice, most handlers need to share data structures with the main program.

[signal trampoline: event loop]

Thread-based Handlers

Many contemporary GUI systems are thread-based. Examples include the **OpenGL** Utility Toolkit (**GLUT**), the **GNU** Image Manipulation Program (**GIMP**) Tool Kit (**Gtk**), the Java Swing library, Windows Forms, and the newer **.NET** Windows Presentation Foundation (**WPF**).

An event handler in C#:



```
void Paused(object sender, EventArgs a) {
    // do whatever needs doing when the pause button is pushed
}
...
Button pauseButton = new Button("pause");
pauseButton.Clicked += new EventHandler(Paused);
```

Thread-based Handlers

An event handler in C#:

```
void Paused(object sender, EventArgs a) {
    // do whatever needs doing when the pause button is pushed
}
...
Button pauseButton = new Button("pause");
pauseButton.Clicked += new EventHandler(Paused);
```

anonymous delegate

```
pauseButton.Clicked += delegate(object sender, EventArgs a) {
    // do whatever needs doing
};
```

Thread-based Handlers

An event handler in Java:

Event Handler

A Listener can Handle Multiple Event

```
class PauseListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // do whatever needs doing
    }
}
...
JButton pauseButton = new JButton("pause");
pauseButton.addActionListener(new PauseListener());
```

Event Source

Connection

Anonymous Inner Class for Event Handler in Java:

```
pauseButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // do whatever needs doing
    }
});
```

Lambda Expression

```
pauseButton.addActionListener(e-> {
// do whatever needs doing
});
```