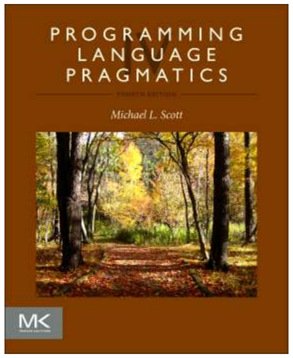
Chapter 9 :: Subroutines and Control Abstraction

- CSE307/526: Principles of Programming Languages
- https://ppawar.github.io/CSE307-F18/index.html

Programming Language Pragmatics, Fourth Edition

Michael L. Scott



Abstractions as program building blocks

- Programming is about building abstractions
- Abstraction allows representing essential features without background details
- Abstraction reduces complexity and allows efficient design and implementation of complex software systems
- The purpose of control abstraction is to perform a well defined operation
- Subroutines are the main method to build control abstractions
- The other form of abstraction we normally think about is data abstraction – OOP (next topic)

Outline

- Introduction to subroutines
- Review of stack layout
- Calling sequences
- Parameter passing
- Generic subroutines and modules
- Exception handling
- Coroutines
- Events

Outline

- Introduction to subroutines
- Review of stack layout
- Calling sequences
- Parameter passing
- Generic subroutines and modules
- Exception handling
- Coroutines
- Events

Subroutines

- Why use subroutines?
  - –Give a name to a task.
  - –We no longer care *how* the task is done.
- The *subroutine call* is an expression
  - –Subroutines take arguments (in the formal parameters)
  - –Values are placed into variables (actual parameters/arguments), and
  - –A value is (usually) returned
- Activation record or stack frames is a means to manage the space for local variables allocated to each subroutine call (Chapter 3)

ELSEVIER

Review Of Memory Layout

- Memory allocation strategies
  - Static (compile time)
    - Code
    - Globals
    - *Own* variables
    - Explicit constants (including strings, sets, other aggregates)
  - Stack (run time)
    - parameters
    - local variables
    - temporaries
    - bookkeeping information
  - Heap
    - dynamic allocation

Recap – Stack based allocation of space for subroutines

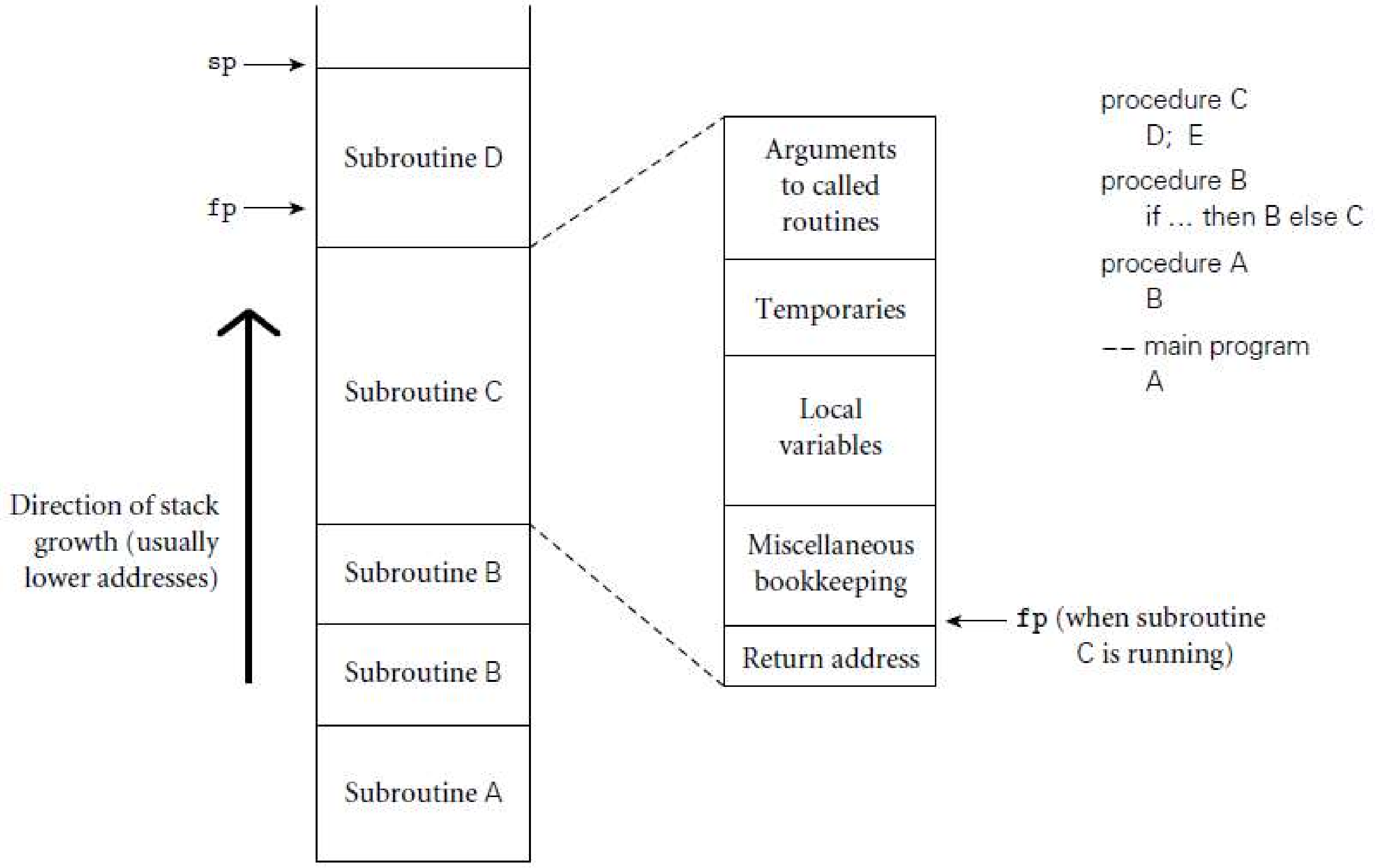


Figure 3.1 **Stack-based allocation of space for subroutines.** We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Recap – Static link

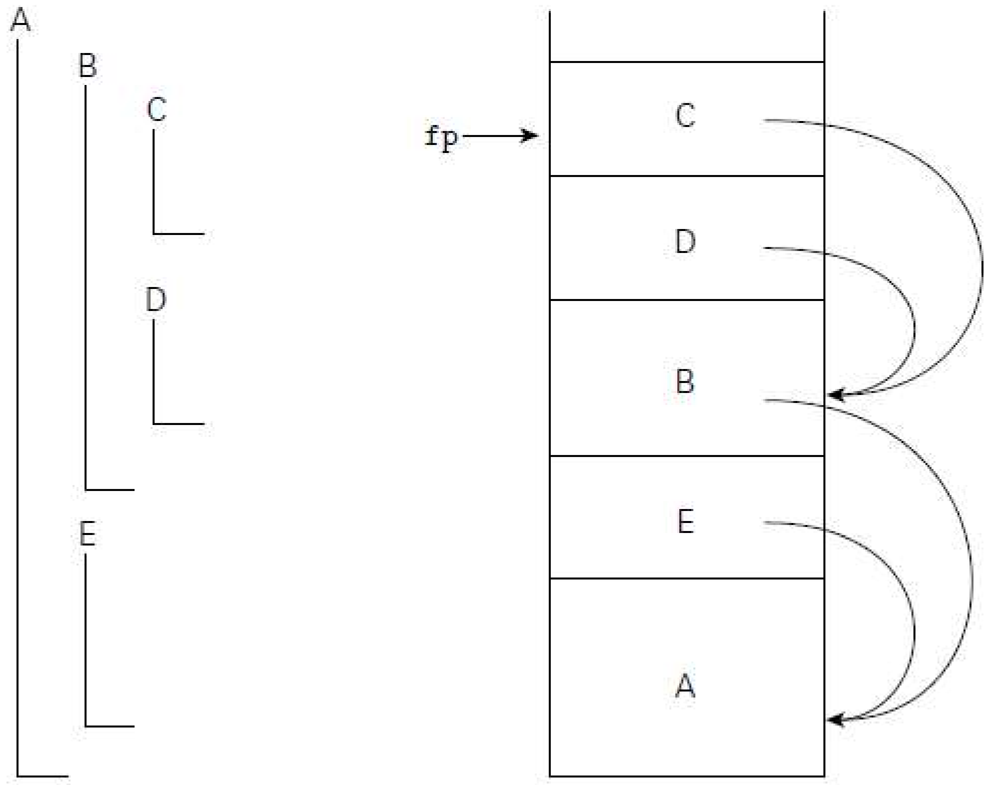


Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

Static links and Dynamic links

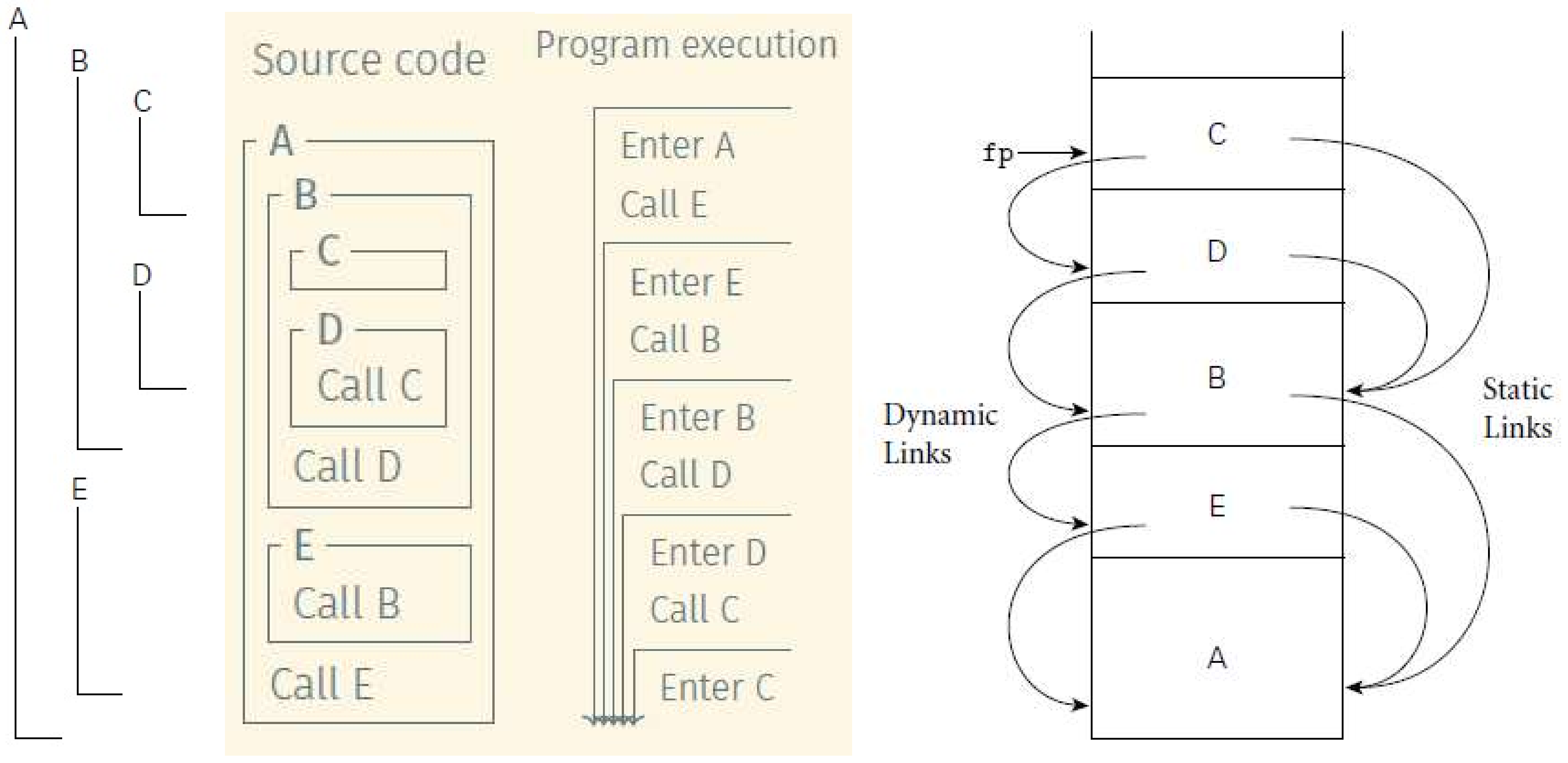


Figure 8.1 Example of subroutine nesting, taken from Figure 3.5. Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.

Calling Sequences

- Maintenance of stack is responsibility of *calling sequence* and *subroutines prolog* and *epilog* – discussed in Chapter 3
  - Tasks on the way to a subroutine: Passing parameters, saving the return address, changing program counter, changing stack pointer to allocate space, saving registers such as frame pointer, changing frame pointer to refer to new frame, executing initialization code for any objects in the new frame
  - Tasks on the way out from a subroutine: Passing return parameters, restoring stack pointer, restoring other registers such as frame pointer, restoring program counter
- space is saved by putting as much in the prolog and epilog as possible
- time *may* be saved by putting stuff in the caller instead

In-line expansion

- Alternative to stack-based calling conventions
- During compile time, the compiler replaces a subroutine call with the code of the subroutine.
- Advantages:
  - –Avoids overhead associated with subroutine calls; faster code.
  - –Encourages building abstractions in the form of many small subroutines.
  - –Related to but cleaner than macros.
- Disadvantages:
  - –Code bloating
  - –Cannot be used for recursive subroutines.
  - –Code profiling becomes more difficult.

Parameter Passing

- Modes of passing parameters:
  - –Pass by value: make a copy of the parameter.
  - –Pass by reference (aliasing): allows the function to change the parameter
  - –Pass by sharing: requires parameter to be a reference itself.
    - •Makes copy of reference that initially refers to the same object.
    - •Within subroutine, value of the object can be changed.
    - •However, identity of the object can not be changed.
    - •E.g., User defined Java Objects.

Parameter Passing

```
def f(a):
  a += 1
x = 0
f(x)
print(x)
```

- value: 0
- reference: 1
- sharing: 0

```
def f(a):
   a.foo = 1
x = object()
x.foo = 0
f(x)
print x.foo
```

- value: 0
- reference: 1
- sharing: 1 (value change)

```
z = object()
z.foo = 1
def f(a):
   a = z
x = object()
x.foo = 0
f(x)
print x.foo
```

- value: 0
- reference: 1
- sharing: 0 (identity change)

ELSEVIER

Parameter Passing

- C/C++ functions
  - parameters passed by value (C)
  - parameters passed by reference can be simulated with pointers (C)

    ```
    void proc(int* x,int y){*x = *x+y } …
    proc(&a,b);
    ```

  - or directly passed by reference (C++)

    ```
    void proc(int& x, int y) {x = x + y }
    proc(a,b);
    ```

- Java
  - Java uses call-by-value for variables of built-in type (all of which are values)
  - Call-by-sharing for variables of user-defined class types (all of which are references).

Parameter Passing – Named Parameters

- The values are passed by ***associating*** each one with a ***parameter name***.
- E.g.,in Objective-C:

```
[window addNewControlWithTitle:@"Title"
        xPosition:20
        yPosition:50
            width:100
           height:50
       drawingNow:YES];
```

- E.g.,in Python:

```
window.addNewControl(title="Title",
                     xPosition=20,
                     yPosition=50,
                     width=100,
                     height=50,
                     drawingNow=true)
```

Parameter Passing – Default Parameters

- **Default parameters**: default values are provided to the function
  - C++ example:

```
void PrintValues(int nValue1, int nValue2=10){
    using namespace std;
    cout << "1st value: " << nValue1 << endl;
    cout << "2nd value: " << nValue2 << endl;
}
int main(){
    PrintValues(1);
    // nValue2 will use default parameter of 10

    PrintValues(3, 4);
    // override default value for nValue2
}
```

Parameter Passing – Variadic functions

- functions of **indefinite** arity – one which accepts variable number of arguments
- Java

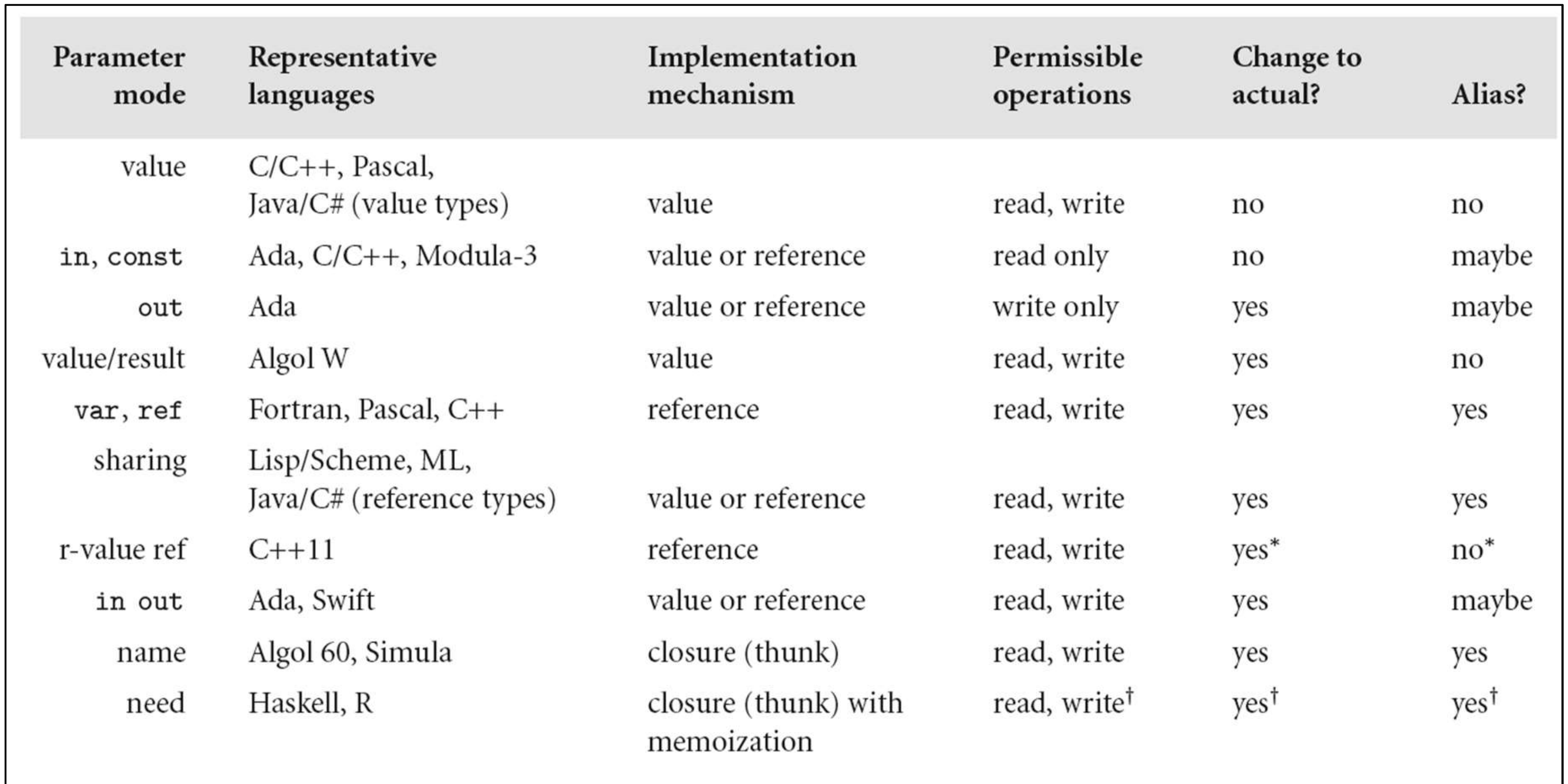
```
public class Program {
    private static void printArgs(String... strings) {
        for (int i = 0; i < strings.length; i++) {
            String string = strings[i];
            System.out.printf("Argument %d: %s", i, string);
        }
    }

    public static void main(String[] args) {
        printArgs("hello", "world");
    }
}
```

Subroutine closures as parameters

- A closure is a reference to subroutine may be passed as a parameter
- Languages that support this:
  - Pascal
  - Ada 95 (not Ada 83)
  - All functional programming languages
- SML example

```
fun apply_to_L(f, l) =
    case l of
        nil => nil
    | h :: t => f(h) :: apply_to_L(f, t);
```

Parameter Passing



Parameter mode	Representative languages	Implementation mechanism	Permissible operations	Change to actual?	Alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
`in`, `const`	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
`out`	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
`var`, `ref`	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
r-value ref	C++11	reference	read, write	yes*	no*
`in out`	Ada, Swift	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write†	yes†	yes†

Figure 9.3 **Parameter-passing modes.** Column 1 indicates common names for modes. Column 2 indicates prominent languages that use the modes, or that introduced them. Column 3 indicates implementation via passing of values, references, or closures. Column 4 indicates whether the callee can read or write the formal parameter. Column 5 indicates whether changes to the formal parameter affect the actual parameter. Column 6 indicates whether changes to the formal or actual parameter, during the execution of the subroutine, may be visible through the other. *Behavior is undefined if the program attempts to use an r-value argument after the call. †Changes to arguments passed by need in R will happen only on the first use; changes in Haskell are not permitted.

Returning from a function

- Different ways of returning a value from a function.
  - Return statement

    return *expression (causes immediate termination of function)*
  - Avoid termination

    rtn := *expression*

    ...

    return rtn
- ML, its descendants, and several scripting languages allow a Multi-value returns
- In Python, for example, we might write

```
def foo():
    return 2, 3
    ...
    i, j = foo()
```

Generic subroutines and modules

- Generic modules or classes are particularly valuable for creating containers: data abstractions that hold a collection of objects
  - When defining a function, we don't need to give all the types
  - When we invoke the class or function we specify the type: *parametric polymorphism*
- Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right

```
public static <T extends Comparable<T>> void sort(T A[]) {
...
if (A[i].compareTo(A[j]) >= 0) {
...
}
...
}
Integer[] myArray = new Integer[50];
sort(myArray);
```

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
- Raising exceptions:
  - Automatically by the run-time system as a result of an abnormal condition
  - (e.g., division by zero)
  - throw/raise statement to raise exceptions manually
  - Most languages allow exceptions to be handled locally and propagate unhandled exceptions up the dynamic chain.

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

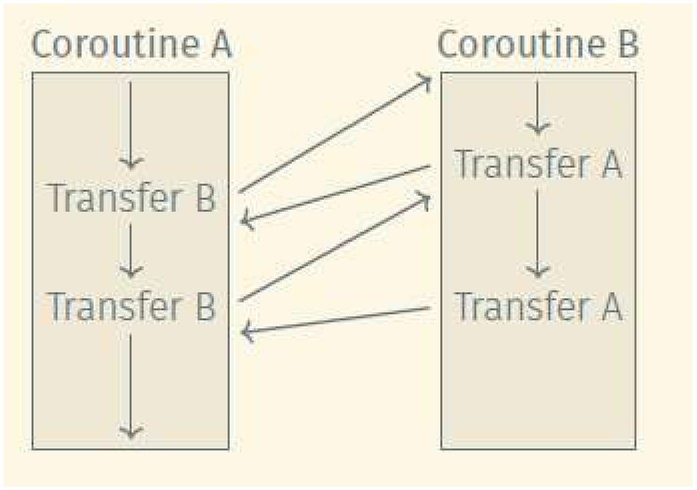
- Java:
  - –throw throws an exception.
  - –try encloses a protected block.
  - –catch defines an exception handler.
  - –finally defines block of clean-up code to execute no matter what.
  - –Only Throwable objects can be thrown.
  - –Must declare uncaught checked exceptions.
- C++:
  - –throw, try, and catch as in Java
  - –No finally block
  - –Any object can be thrown.
  - –Exception declarations on functions not required

```
try {
...
throw ...
...
}
catch (SomeException e1) {
...
}
catch (SomeException e2) {
...
}
finally {
...
}
```

ELSEVIER

Coroutines

- Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other voluntarily and explicitly, by name
- Coroutines can be used to implement
  - iterators
  - threads
  - Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack
  - But they may use static link for scoping



Event types

- 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
- A handler—a special subroutine— is invoked when a given event occurs.
- Thread-Based Handlers:
  - –In modern programming languages and run-time systems, events are often handled by a separate thread of control, rather than by spontaneous subroutine calls

Event types

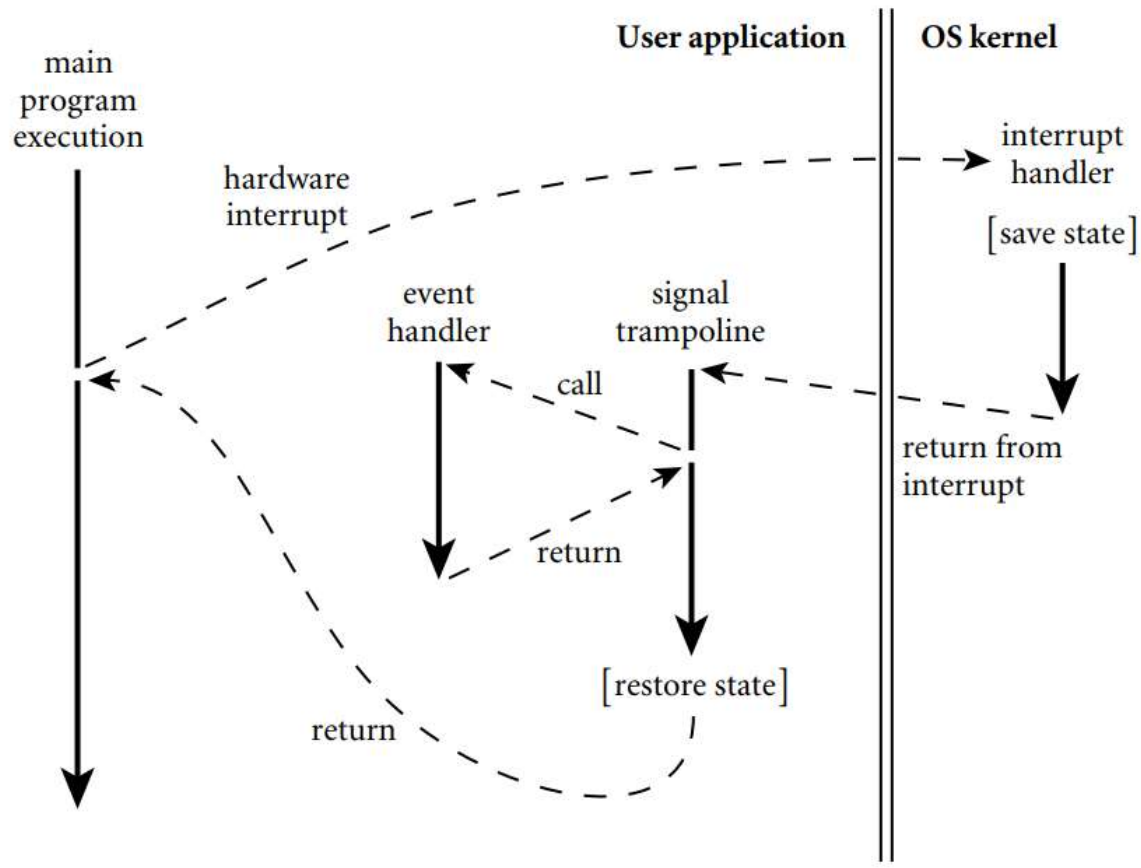


Figure 8.7 **Signal delivery through a trampoline.** When an interrupt occurs (or when another process performs an operation that should appear as an event), the main program may be at an arbitrary place in its code. The kernel saves state and invokes a *trampoline* routine that in turn calls the event handler through the normal calling sequence. After the event handler returns, the trampoline restores the saved state and returns to where the main program left off.