Control Flow

Chapter 6

Control Flow

- Skipping most of this chapter in class
  - Skim it when reading
  - We'll just hit on a few topics that may be new
- Basic paradigms for control flow:
  - Sequencing
  - Selection
  - Iteration and Recursion
  - Procedural Abstraction
  - Concurrency
  - Nondeterminacy

Precedence Rules

- Varies from language to language; the smart programmer always uses parentheses where questionable to ensure proper precedence

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /= , <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if... then ... else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Figure 6.1: **Operator precedence levels in Fortran, Pascal, C, and Ada.** The operators at the top of the figure group most tightly.

Concepts in Expression Evaluation

- Ordering of operand evaluation (generally none)
- Application of arithmetic identities
  - commutativity (assumed to be safe)
  - associativity (known to be dangerous)

  `(a + b) + c` works if `a~=maxint` and `b~=minint` and `c<0`

  `a + (b + c)` does not
- Short Circuit
- Variables as values vs. variables as references
- Orthogonality
  - Features that can be used in any combination (Algol 68)

```
a:= if b < c then d else e;       //  "returns" a val
a:= begin f(b); g(c); end;        //  "returns" a val
```

Concepts in Expression Evaluation

- Side Effects
  - often discussed in the context of functions
  - a side effect is some permanent state change caused by execution of function
    - some noticeable effect of call other than return value
    - in a more general sense, assignment statements provide the ultimate example of side effects
      - they change the value of a variable
- Side Effects are fundamental to the von Neumann computing model
- In (pure) functional, logic, and dataflow languages, there are no such changes
  - These languages are called SINGLE-ASSIGNMENT languages
  - Expressions in a purely functional language are REFERENTIALLY TRANSPARENT

Concepts in Expression Evaluation

- Boxing
  - A drawback of using the primitive value model for built-in types is they can't be passed to methods that expect references to objects
  - Boxing is "wrapping" a primitive in an object
  - Early Java: wrap explicitly using Integer, Double, etc.
  - Later Java: Automatic boxing and unboxing

```
Integer x = 6;            //6 is boxed
Integer y = 2*x + 3;      //x is unboxed, 15 is boxed
```

Concepts in Expression Evaluation

- Initialization
  - Definite assignment: variable must have a value assigned before being used
- Constructors
- Combination assignment, multiway assignment

Structured Flow

- Sequencing
  - specifies a linear ordering on statements
    - one statement follows another
  - very imperative, Von-Neumann
- Selection
  - sequential if statements or switch/select statements

```
if ... then ... else
if ... then ... elsif ... else
(cond
        (C1) (E1)
        (C2) (E2)
        ...
        (Cn) (En)
        (T)  (Et)
)
```

GOTO Statement

- Once the primary means of control flow

```
10  A = 1
11  PRINT "HELLO"
12  A = A + 1
13  IF A < 100 GOTO 11
14  PRINT "DONE"
```

- Heavily discussed in the 60's and 70's
  - Dijkstra – GOTO's Considered Harmful
- Mostly abandoned in favor of **structured programming**; some languages allow labels (can be useful for immediate exit)
  - Not allowed at all in Java; can't even use the keyword

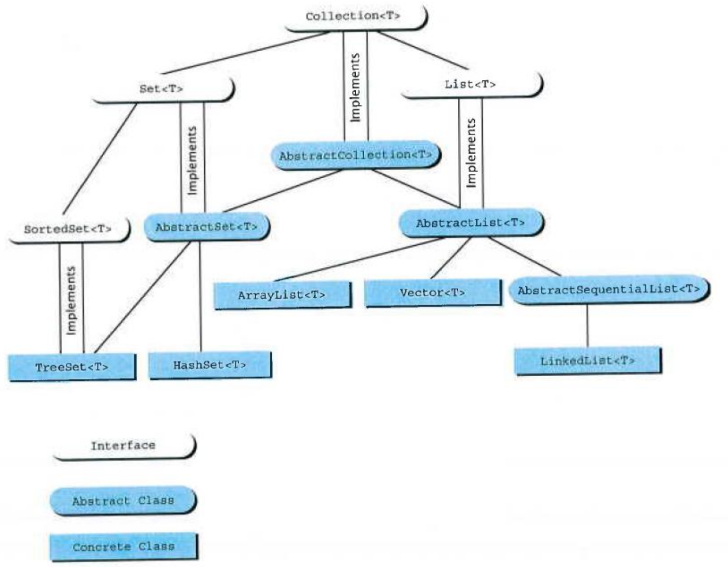
Iteration

- Loops
  - While, repeat, for
  - foreach
- Recursion
  - Tail recursion
  - No computation follows recursive call

```
/* assume a, b > 0 */
int gcd (int a, int b) {
   if (a == b) return a;
   else if (a > b) return gcd (a - b, b);
   else return gcd (a, b – a);
}
```

- Iterators
  - In general we may wish to iterate over the elements of any well-defined set (container or collection)
    - ArrayList, Set (maybe), HashTable, etc.

Java Collections



Java Iterators

- Iterator used with a collection to provide sequential access to the elements in the collection
- Methods in the Iterator<T> Interface

The `Iterator<T>` interface is in the `java.util` package.

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

`NoSuchElementException` is in the `java.util` package, which requires an import statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any import statement.

```
public T next( )
```

Returns the next element of the collection that produced the iterator.

Throws a `NoSuchElementException` if there is no next element.

```
public boolean hasNext( )
```

Returns `true` if `next( )` has not yet returned all the elements in the collection; returns `false` otherwise.

```
public void remove( ) (Optional)
```

Removes from the collection the last element returned by `next`.

```
public class HashSetIteratorDemo
{
    public static void main(String[] args)
    {
        HashSet<String> s = new HashSet<String>( );

        s.add("health");
        s.add("love");
        s.add("money");

        System.out.println("The set contains:");

        Iterator<String> i = s.iterator( );
        while (i.hasNext( ))
            System.out.println(i.next( ));

        i.remove( );

        System.out.println( );
        System.out.println("The set now contains:");

        i = s.iterator( );
        while (i.hasNext( ))
            System.out.println(i.next( ));

        System.out.println("End of program.");
    }
}
```

You cannot "reset" an iterator "to the beginning." To do a second iteration, you create another iterator.

Sample Dialogue

```
The set contains:
money
love
health

The set now contains:
money
love
End of program.
```

The **HashSet<T>** *object does not order the elements it contains, but the iterator imposes an order on the elements.*

Applicative and Normal-Order Evaluation

- We normally assume all arguments are evaluated before passing them to a subroutine
  - Required for many languages so we know what to stick on the stack
  - But this need not always be the case
- Applicative Order Evaluation
  - Evaluating all arguments before the call
- Normal Order Evaluation
  - Evaluating only when the value is actually needed
  - Used in some functional languages

Normal Order Evaluation

- Applicative Order is the norm
- Normal Order can sometimes lead to faster code, code that works when applicative-order evaluation would cause a run-time error
  - E.g. short circuit
- Lazy Evaluation in Scheme
  - Built-in functions delay and force
  - Keeps track of expressions already evaluated so it can reuse their values if they are needed more than once
    - Also called memoization
    - E.g.: fib(n) { if (n<=2) return n else return fib(n-1)+fib(n-2) }