Iterators are classified along two dimensions.

**Location of iteration control**
- Control is *external* to iterator, i.e. iterator client has control
- Control is *internal* to iterator, i.e. iterator has control

**Location of iterator**
- Iterator is *embedded* in the collection object
- Iterator is a *separate* class from the collection class
One iterator characteristic is the location of the iterator class with respect to the collection class.

- The iterator definition can be in a class separate from the collection.
  - Good separation of concerns: traversal vs. maintenance of collection structure
  - Multiple traversal types (forward, backward, matching) without unneeded ones
  - Easier to do multiple traversals at the same time

- The iterator definition can be embedded within the collection class.
  - Preserves encapsulation of collection

Who controls iteration when you use the Java Collections Framework iterators?

- What is the programming pattern for using the Java Collection framework iterators?

  Iterator iter = theCollection.iterator();

  while( iter.hasNext() ) {
      process( iter.next() );
  }

  This is external control.

  Is the iterator separate or embedded?
Internal iterators require the client to provide a processing function.

- Iterator has a function to perform iteration

```java
public interface InternalIterator {
    public boolean iterate(Processor p);
}
```

- Client requests that the iterator iterate through the collection and process each object

```java
public interface Processor {
    public boolean process(Object o);
}
```

The client has no iteration control with an internal iterator except to possibly terminate early.

```java
public class ACollectionIterator implements InternalIterator {
    private SomeCollection theCollection;
    public ACollectionIterator(SomeCollection c) {
        theCollection = c;
    }
    public boolean iterate(Processor p) {
        boolean result = true;
        Start at the beginning of theCollection
        while (still elements && result) {
            if (!p.process(next element)) {
                result = false;
            }
            Move to the next element
        }
        return result;
    }
}
```
Internal Iterators in Ruby

Standard Ruby container classes use an application of internal iterators:

```ruby
a = [ 10, 20, 30 ]
a.each { |element| puts( "The element is #{element}" ) }
```

outputs:
The element is 10
The element is 20
The element is 30

The `each` method executes a {code block} on every element of the array object – **control is internal, iterator is embedded**.

Arrays – `each`, `each_index`, `reverse_each`  
Hash – `each`, `each_pair`, `each_key`, `each_value`

---

External Iterators in Ruby

IO class external iterator

```ruby
f = File.open('names.txt')
while not f.eof?
  puts(f.readline)
end
f.close
```

**control is external, iterator is embedded**

Performing the same iteration internally:

```ruby
f = File.open('names.txt')
f.each { |line| puts(line) }
f.close
```

also:

```ruby
f.each_byte { |byte| puts(byte) }
```

**control is internal, iterator is embedded**
Implementing internal iterators in Ruby

You can add similar behavior in your own classes by implementing the each method and gain additional functionality by including the Enumerable mix-in module.

```ruby
class Portfolio
  include Enumerable
  def initialize
    @accounts = []
  end
  def each(&block)
    @accounts.each(&block)
  end
  def add_account(account)       # Note: definition of account class not shown here
    @accounts << account
  end
end
```

You can then iterate across Account objects and execute the code block passed in as `&block`. In this case Portfolio is simply using the each method for the array class, but you could supply your own iterator for more complex composites.

Enumerable methods use your each method to do things like: any?, find, grep, include?, max, member?, min, sort

```ruby
#example – Do any accounts in the portfolio have a balance of a least $2000?
my_portfolio.any? { |account| account.balance > 2000 }
```

There are advantages and disadvantages with external and internal iterators.

- **External iterators** are more flexible
  - *Example: Compare two lists for equality*

- **Internal iterators** are easier to use
  - *Iteration logic handled for client*
Placement of knowledge of how to do traversal involves trade-offs of O-O design principles.

- **Iterator determines next element**
  - Here iterator needs to know structure of collection
  - Violates encapsulation of collection if a separate iterator

- **Collection determines next element**
  - Iterator is just a marker/cursor of where traversal left off
  - Traversal logic is in the collection even if a separate iterator
  - Complicates collection itself
  - Does not separate iteration from collection maintenance

Recursive collections, such as Composites, present special problems for iteration.

- **Is programming pattern for Java Collections framework iterative or recursive?**

- **What is the easiest way to traverse a recursive collection like a tree?**

- **Can you traverse it with the other approach?**
  - What are the issues?
It is easier to iterate through a recursive collection with an internal iterator.

\[\text{External iterator}\]
- iterator stores path for retreat (Hansel & Gretel)
- composite provides back links for iterator to use (parent / sibling / children)

\[\text{Internal iterator}\]
- Traverse recursively – get backtrack for free
- Get iterator for children of current element
- Use null iterators at leaves.