Iterator

Baris Aktemur

CS 534| Ozyegin University

Contents are from “Design Patterns” by Gamma, Helm, Johnson, Vlissides

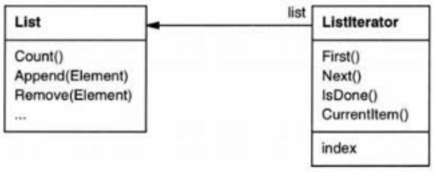
Iterator

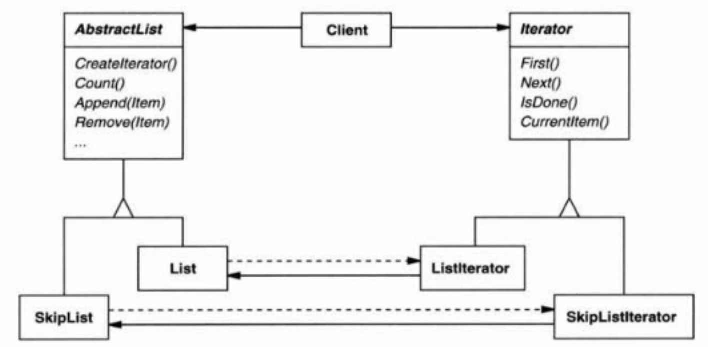
- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Motivation
  - access elements of a list without exposing its internal structure
  - traverse the list in different ways, but without bloating the List interface with operations for different traversals

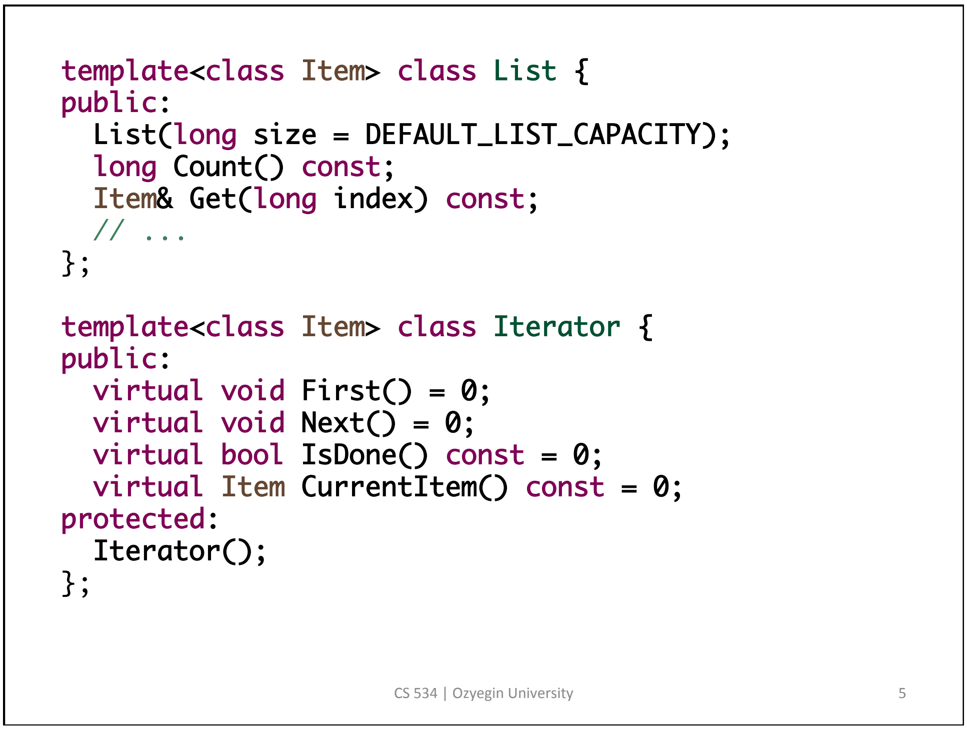
Iterator

- Takes the responsibility for access and traversal out of the list object and puts it into an iterator object.
- The Iterator class defines an interface for accessing the list's elements.
- An iterator object is responsible for keeping track of the current element; it knows which elements have been traversed already.

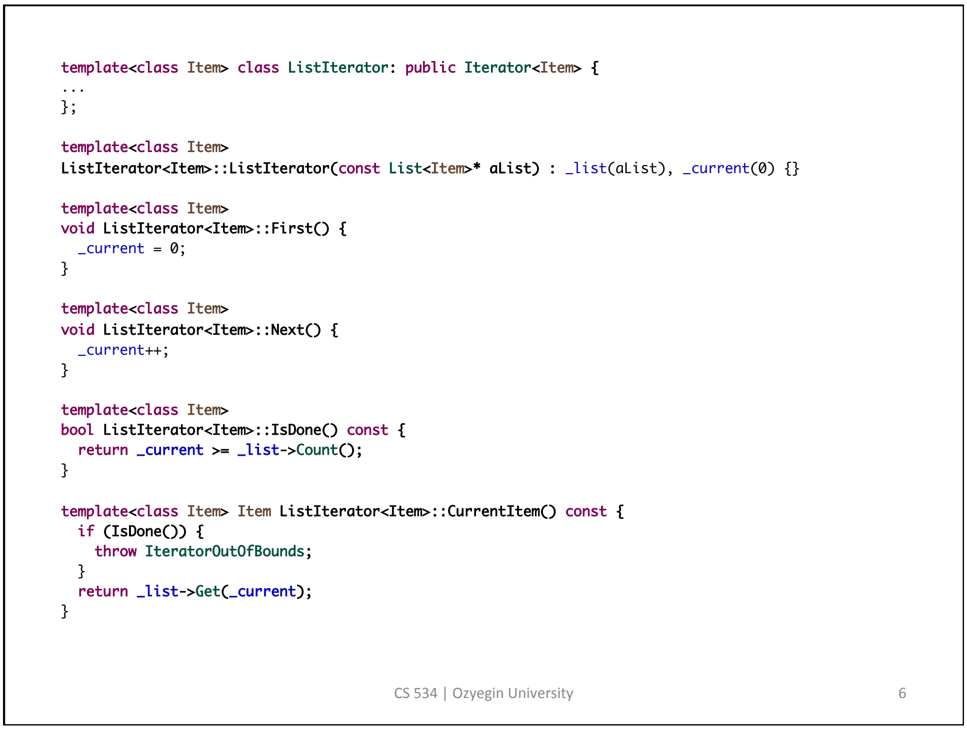
Iterator



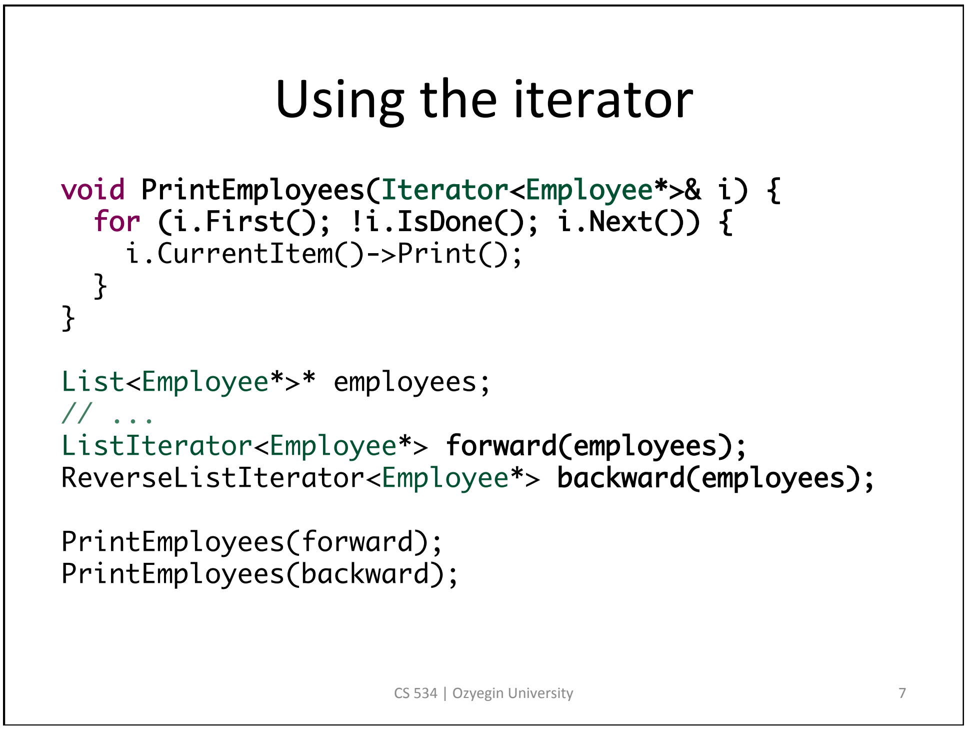




```
template<class Item> class List {
public:
  List(long size = DEFAULT_LIST_CAPACITY);
  long Count() const;
  Item& Get(long index) const;
  // ...
};

template<class Item> class Iterator {
public:
  virtual void First() = 0;
  virtual void Next() = 0;
  virtual bool IsDone() const = 0;
  virtual Item CurrentItem() const = 0;
protected:
  Iterator();
};
```

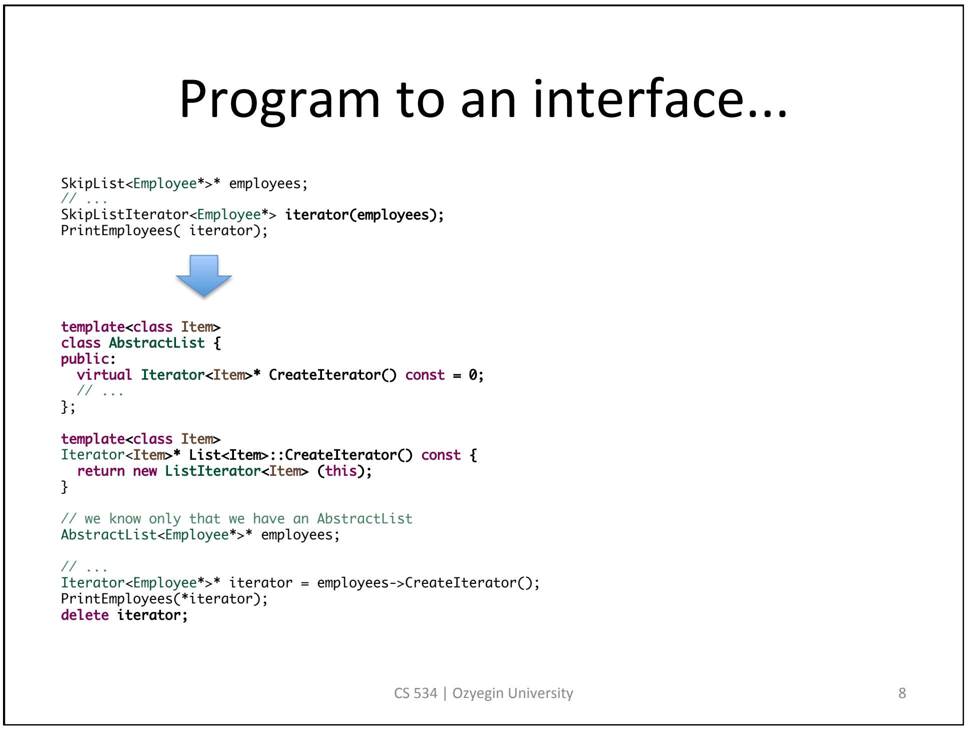


```
template<class Item> class ListIterator: public Iterator<Item> {
...
};

template<class Item>
ListIterator<Item>::ListIterator(const List<Item>* aList) : _list(aList), _current(0) {}

template<class Item>
void ListIterator<Item>::First() {
  _current = 0;
}

template<class Item>
void ListIterator<Item>::Next() {
  _current++;
}

template<class Item>
bool ListIterator<Item>::IsDone() const {
  return _current >= _list->Count();
}

template<class Item> Item ListIterator<Item>::CurrentItem() const {
  if (IsDone()) {
    throw IteratorOutOfBounds;
  }
  return _list->Get(_current);
}
```



Using the iterator

```
void PrintEmployees(Iterator<Employee*>& i) {
  for (i.First(); !i.IsDone(); i.Next()) {
    i.CurrentItem()->Print();
  }
}

List<Employee*>* employees;
// ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```



Program to an interface...

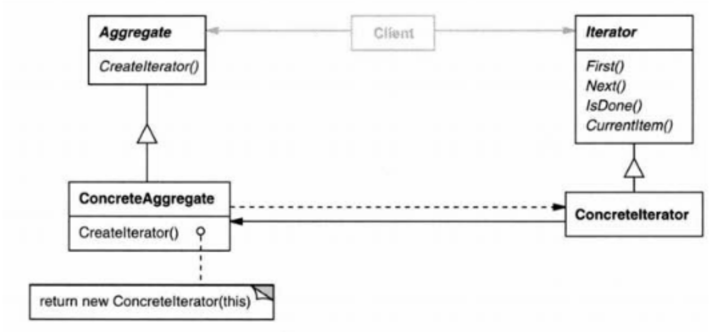
```
SkipList<Employee*>* employees;
// ...
SkipListIterator<Employee*> iterator(employees);
PrintEmployees( iterator);
```

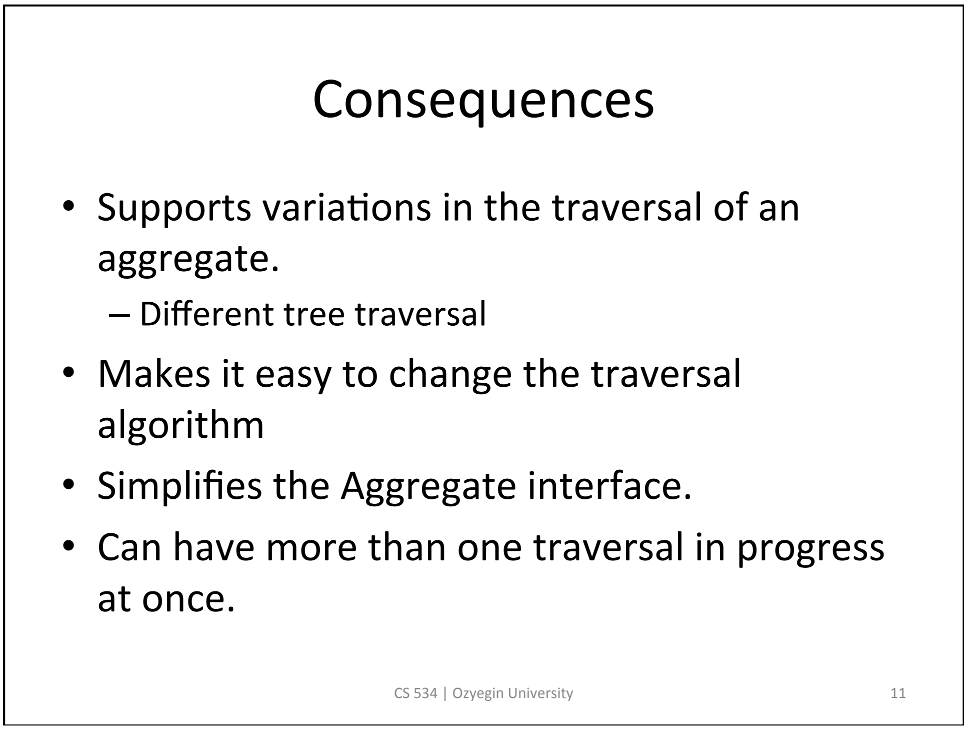
```
template<class Item>
class AbstractList {
public:
  virtual Iterator<Item>* CreateIterator() const = 0;
  // ...
};

template<class Item>
Iterator<Item>* List<Item>::CreateIterator() const {
  return new ListIterator<Item> (this);
}

// we know only that we have an AbstractList
AbstractList<Employee*>* employees;

// ...
Iterator<Employee*>* iterator = employees->CreateIterator();
PrintEmployees(*iterator);
delete iterator;
```

Iterator

- CreateIterator: Isolate creation of the iterator.
  - A **Factory Method**.

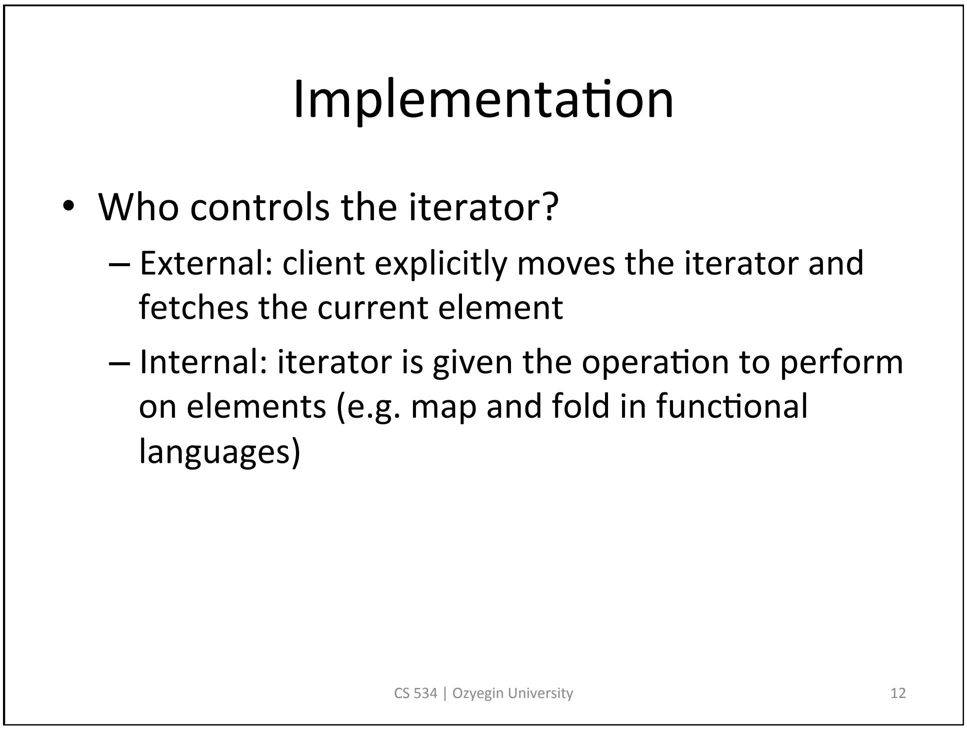
Structure





Consequences

- Supports variations in the traversal of an aggregate.
  - Different tree traversal
- Makes it easy to change the traversal algorithm
- Simplifies the Aggregate interface.
- Can have more than one traversal in progress at once.



Implementation

- Who controls the iterator?
  - External: client explicitly moves the iterator and fetches the current element
  - Internal: iterator is given the operation to perform on elements (e.g. map and fold in functional languages)

Internal Iterator

```
template<class Item>
class ListTraverser {
public:
  ListTraverser(List<Item>* aList);
  bool Traverse();
protected:
  virtual bool ProcessItem(const Item&) = 0;
private:
  ListIterator<Item> _iterator;
};

template<class Item>
ListTraverser<Item>::ListTraverser(List<Item>* aList) : _iterator(aList) {}

template<class Item>
bool ListTraverser<Item>::Traverse(){
  bool result = false;
  for ( _iterator.First(); !_iterator.IsDone(); _iterator.Next()) {
      result = ProcessItem(_iterator.CurrentItem());
      if (result == false) {
          break;
      }
  }
  return result;
}
```

Print 10 Employees

```
class PrintNEmployees: public ListTraverser<Employee*> {
public:
  PrintNEmployees(List<Employee*>* aList, int n) :
    ListTraverser<Employee*> (aList), _total(n), _count(0) {}

protected:
  bool ProcessItem(Employee* const &);
private:
  int _total;
  int _count;
};

bool PrintNEmployees::ProcessItem(Employee* const & e) {
  _count++;
  e->Print();
  return _count < _total;
}

List<Employee*>* employees;
// ...
PrintNEmployees pa(employees, 10);
pa.Traverse();
```

Print 10 Employees

```
List<Employee*>* employees;
// ...
PrintNEmployees pa(employees, 10);
pa.Traverse();
```

⇕

```
ListIterator<Employee*> i(employees);
int count = 0;
for (i.First(); !i.IsDone(); i.Next()) {
  count++;
  i.CurrentItem()->Print();
  if (count >= 10) {
    break;
  }
}
```

Another Internal Iterator

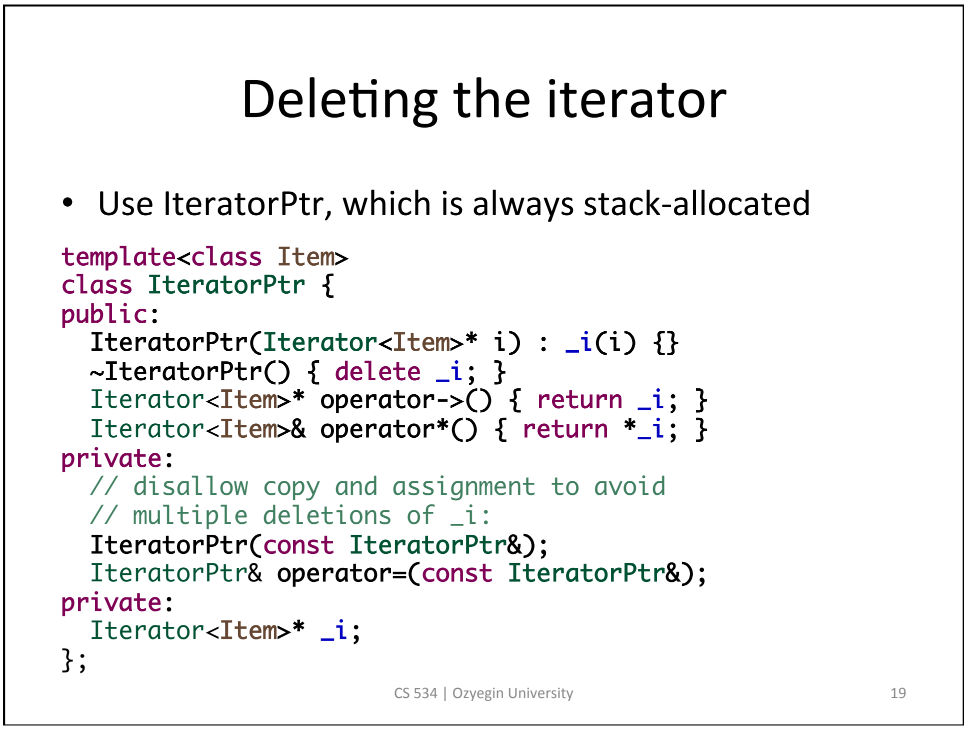
```
template<class Item> class FilteringListTraverser {
public:
  FilteringListTraverser(List<Item>* aList);
  bool Traverse();
protected:
  virtual bool ProcessItem(const Item&) = 0;
  virtual bool TestItem(const Item&) = 0;
private:
  ListIterator<Item> _iterator;
};

template<class Item> void FilteringListTraverser<Item>::Traverse() {
  bool result = false;
  for (_iterator.First(); !_iterator.IsDone(); _iterator.Next()) {
    if (TestItem(_iterator.CurrentItem())) {
      result = ProcessItem(_iterator.CurrentItem());
      if (result == false) {
        break;
      }
      return result;
    }
  }
}
```

Implementation

- Who defines the traversal algorithm?
  - Client: iterator is merely a pointer. “next” operation is called on the client, which sets the new value of the iterator.
  - Iterator: Allows using different algorithms. If the iterator needs to access private fields of the aggregate, encapsulation will be broken.
- How robust is the iterator?
  - Adding/removing elements while traversing?

Implementation

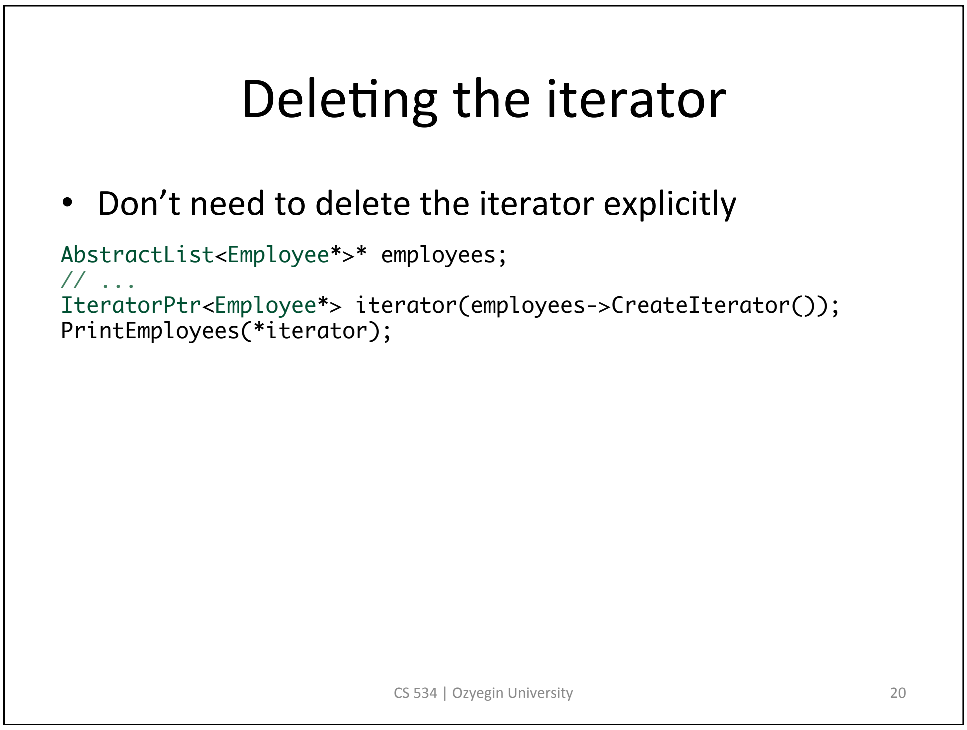
- Who deletes the iterator?
  - Concrete iterators: Can be stack-allocated. Automatic deallocation.
  - Polymorphic iterators: Heap-allocated. Explicit management.
- Iterators for composites
  - Tricky. Composite interface should provide traversal methods to have an external iterator.
- NullIterator
  - A Leaf element does not have any children. Returns a NullIterator for uniformity.
  - See the NullObject pattern.



Deleting the iterator

- Use IteratorPtr, which is always stack-allocated

```
template<class Item>
class IteratorPtr {
public:
  IteratorPtr(Iterator<Item>* i) : _i(i) {}
  ~IteratorPtr() { delete _i; }
  Iterator<Item>* operator->() { return _i; }
  Iterator<Item>& operator*() { return *_i; }
private:
  // disallow copy and assignment to avoid
  // multiple deletions of _i:
  IteratorPtr(const IteratorPtr&);
  IteratorPtr& operator=(const IteratorPtr&);
private:
  Iterator<Item>* _i;
};
```



Deleting the iterator

- Don’t need to delete the iterator explicitly

```
AbstractList<Employee*>* employees;
// ...
IteratorPtr<Employee*> iterator(employees->CreateIterator());
PrintEmployees(*iterator);
```