CS 525 - ASD
Advanced Software Development

MS.CS Program

Department of Computer Science

Rene de Jong, MsC.

Maharishi University
OF MANAGEMENT

CS 525 - ASD

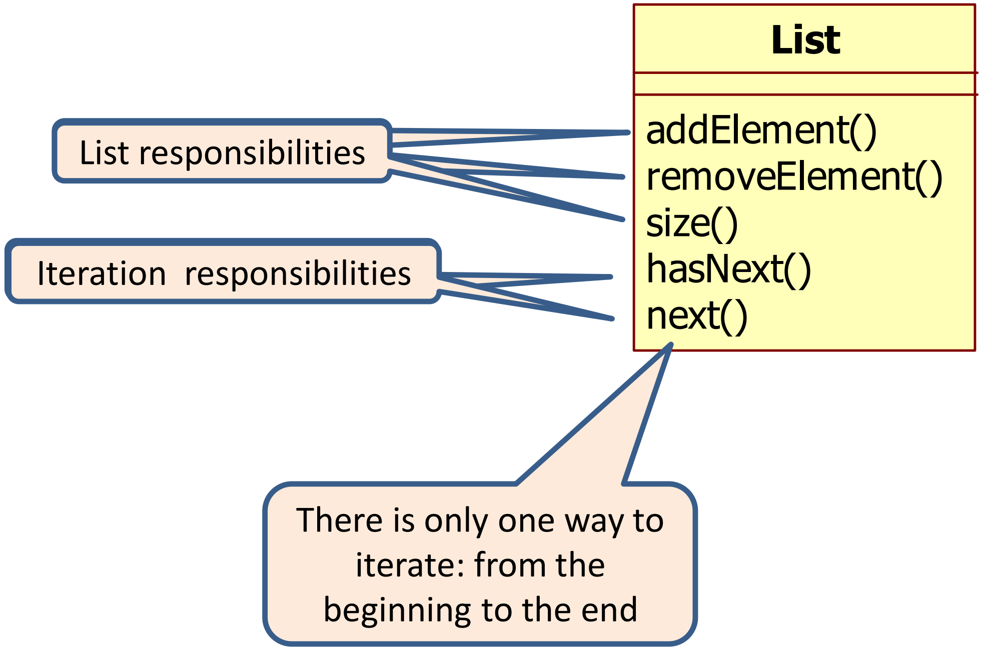
Advanced Software Development

Maharishi University
OF MANAGEMENT

Iterator pattern

- Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying implementation.
- An iterator object encapsulates the internal structure of how the iteration occurs.

Without the Iterator pattern

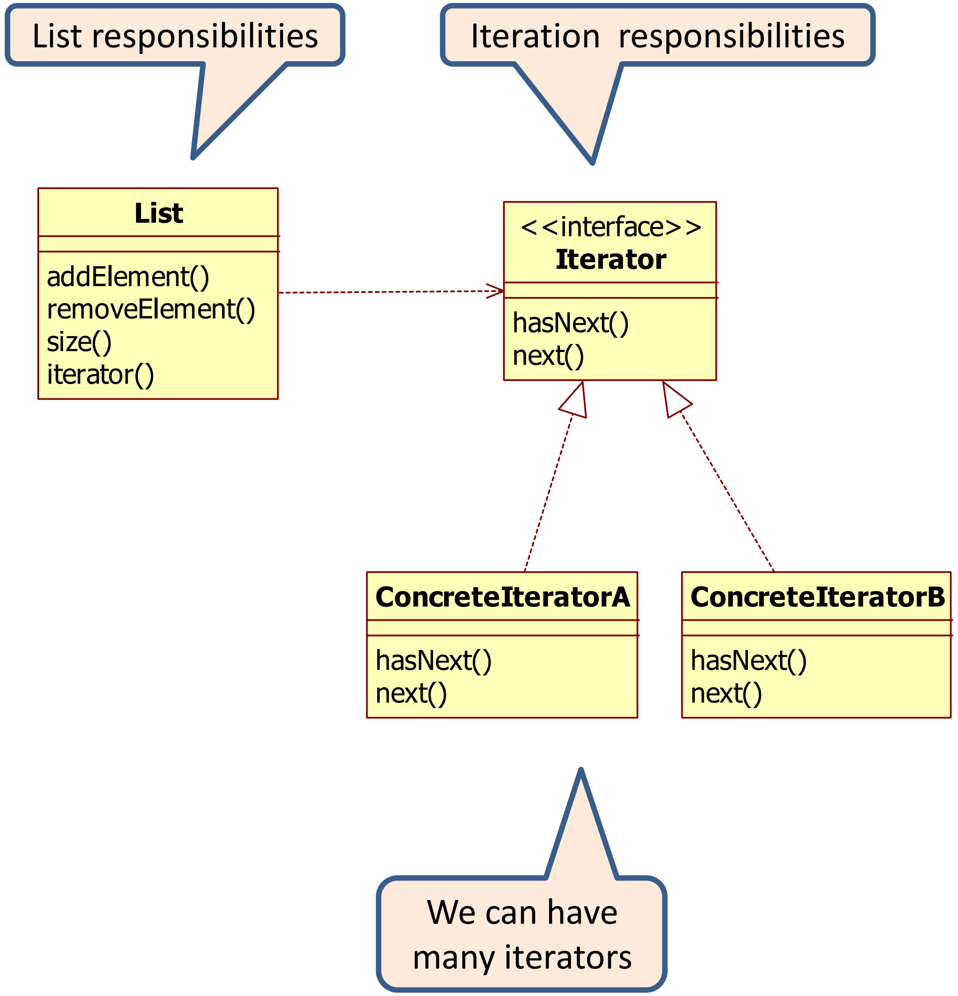


List responsibilities

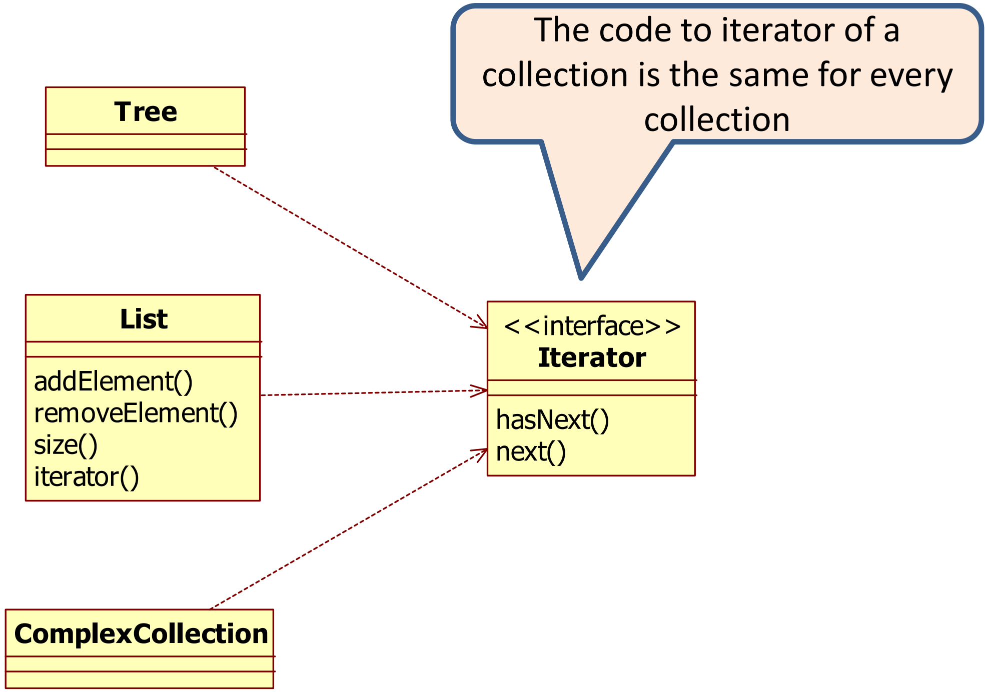
Iteration responsibilities

There is only one way to iterate: from the beginning to the end

With the Iterator pattern



With the Iterator pattern



Iterators

- External iterator
  - The client controls the iteration
- Internal iterator
  - The iterator controls the iteration

External iteration in Java

```java
public class ApplicationForEach {

  public static void main(String[] args){
    List<String> alphabet = new ArrayList<String>();
    alphabet.add("a");
    alphabet.add("b");
    alphabet.add("c");

    Iterator<String> iterator = alphabet.listIterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.next().toUpperCase());
    }
  }
}
```

External iterator

Enhanced for loop iteration in Java

```
public class ApplicationForEach {

  public static void main(String[] args){
    List<String> alphabet = new ArrayList<String>();
    alphabet.add("a");
    alphabet.add("b");
    alphabet.add("c");

    for(String letter: alphabet){
      System.out.println(letter.toUpperCase());
    }
  }
}
```

The underlying code which makes this iteration work uses an external iterator and calls next() and hasNext() methods

Internal iteration in Java

```
public class ApplicationInternalIterator {

  public static void main(String[] args) {
    List<String> alphabet = new ArrayList<String>();
    alphabet.add("a");
    alphabet.add("b");
    alphabet.add("c");

    alphabet.forEach(l -> System.out.println(l.toUpperCase()));
  }
}
```

Internal iterator

Removing an element from a collection

```
public class ApplicationForEachException {

  public static void main(String[] args){
    List<String> alphabet = new ArrayList<String>();
    alphabet.add("a");
    alphabet.add("b");
    alphabet.add("c");

    for(String letter: alphabet){
      if (letter.equals("c"))
        alphabet.remove(letter);
    }
  }
}
```

ConcurrentModificationException

```
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
at java.util.ArrayList$Itr.next(ArrayList.java:859)
at removing.with.iterator.ApplicationForEachException.main(ApplicationForEachException.java:15)
```

Removing an element from a collection

```java
public class ApplicationInternalIterator {

  public static void main(String[] args) {
    List<String> alphabet = new ArrayList<String>();
    alphabet.add("a");
    alphabet.add("b");
    alphabet.add("c");

    alphabet.forEach(l -> {if (l.equals("c"))   alphabet.remove(l);});
    alphabet.forEach(l -> System.out.println(l.toUpperCase()));
  }
}
```

ConcurrentModificationException

```
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.ArrayList.forEach(ArrayList.java:1260)
at removing.with.iterator.ApplicationInternalIterator.main(ApplicationInternalIterator.java:15)
```

Removing an element from a collection

```
public class ApplicationForEachSuccess {

  public static void main(String[] args){
    String toBeRemoved = null;
    List<String> alphabet = new ArrayList<String>();
    alphabet.add("a");
    alphabet.add("b");
    alphabet.add("c");

    for(String letter: alphabet){
      if (letter.equals("c"))
        toBeRemoved=letter;
    }
    alphabet.remove(toBeRemoved);

    for(String letter: alphabet){
      System.out.println(letter.toUpperCase());
    }

  }
}
```

Call remove() outside the loop

Removing an element from a collection

```java
public class ApplicationExternalIterator {

  public static void main(String[] args) {
    List<String> alphabet = new ArrayList<String>();
    alphabet.add("a");
    alphabet.add("b");
    alphabet.add("c");

    Iterator<String> iterator = alphabet.listIterator();
    while (iterator.hasNext()) {
      String element = iterator.next();
      if (element.equals("c"))
        iterator.remove();
    }

    iterator = alphabet.listIterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.next().toUpperCase());
    }
  }
}
```

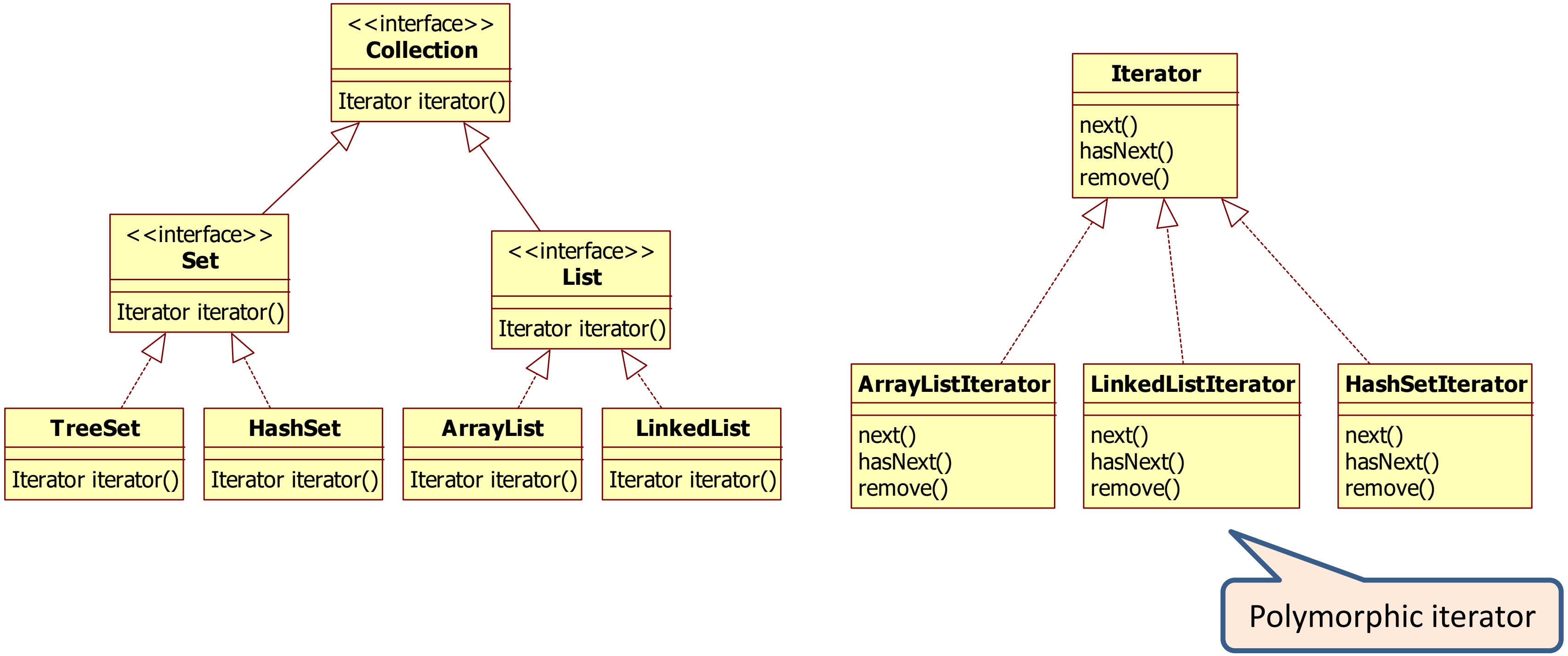
Call remove() on the iterator

Removing an element from a collection

```
public class ApplicationInternalIteratorSuccess {

  public static void main(String[] args) {
    List<String> alphabet = new ArrayList<String>();
    alphabet.add("a");
    alphabet.add("b");
    alphabet.add("c");

    alphabet.removeIf(l -> l.equals("c"));

    alphabet.forEach(l -> System.out.println(l.toUpperCase()));
  }
}
```

removeIf() uses the internal iterator

Iterator in Java collection framework



Polymorphic iterator

Iterator in Java collection framework

```java
public class Application {
  public static void main(String[] args) {
    Collection<String> col1 = new ArrayList<>();
    col1.add("a");
    col1.add("b");
    col1.add("c");
    Collection<String> col2 = new HashSet<>();
    col1.add("a");
    col1.add("b");
    col1.add("c");
    Collection<String> col3 = new LinkedList<>();
    col1.add("a");
    col1.add("b");
    col1.add("c");
    printCollection(col1);
    printCollection(col2);
    printCollection(col3);
  }

  public static void printCollection(Collection<String> collection) {
    Iterator<String> iterator = collection.iterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.next());
    }
  }

}
```

Polymorphic iterator

Writing your own iterator

```
public class ReverseIterator<T> implements Iterator<T>{
  private final List<T> list;
  private int position;

  public ReverseIterator(List<T> list) {
    this.list = list;
    this.position = list.size() - 1;
  }

  public Iterator<T> iterator() {
    return this;
  }

  @Override
  public boolean hasNext() {
    return position >= 0;
  }

  @Override
  public T next() {
    return list.get(position--);
  }

  @Override
  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

Implement the Iterator interface

Iterate from the back to the front of the list

Not supported

Using your own iterator

```
public class ProductCollection {
  private List<Product> products = new ArrayList<>();

  public void addProduct(Product product){
    products.add(product);
  }

  public Iterator<Product> reverseIterator(){
    return new ReverseIterator<Product>(products);
  }
}
```

Factory method creates the iterator

```
public class Product {
  private String number;
  private String name;
  private double price;
  private boolean available;
  ...
}
```

Using your own iterator

```java
public class Application {

  public static void main(String[] args) {
    ProductCollection productCollection = new ProductCollection();
    productCollection.addProduct(new Product("A234", "Iphone 10", 850.0, true));
    productCollection.addProduct(new Product("A235", "Iphone 11", 1050.0, false));
    productCollection.addProduct(new Product("A236", "Iphone 9", 650.0, true));
    productCollection.addProduct(new Product("A238", "Iphone 8", 425.0, true));


    Iterator<Product> reverseIterator = productCollection.reverseIterator();
    while (reverseIterator.hasNext()) {
      System.out.println(reverseIterator.next());
    }
  }
}
```

```
Product [number=A238, name=Iphone 8, price=425.0, available=true]
Product [number=A236, name=Iphone 9, price=650.0, available=true]
Product [number=A235, name=Iphone 11, price=1050.0, available=false]
Product [number=A234, name=Iphone 10, price=850.0, available=true]
```

Writing your own iterator with a filter

```
public class FilterIterator<T> implements Iterator<T>{
  private final List<T> list;
  private int position;
  private Predicate<T> predicate;

  public FilterIterator(List<T> list, Predicate<T> predicate) {
    this.list = list;
    this.predicate=predicate;
    this.position = 0;
  }

  public Iterator<T> iterator() {
    return this;
  }

  @Override
  public boolean hasNext() {
    int tempPosition = position;
    while (tempPosition < list.size()) {
      T nextElement = list.get(tempPosition);
      if (predicate.test(nextElement)) {
        return true;
      }
      else {
        tempPosition++;
      }
    }
    return false;
  }
```

Pass a predicate

See if there is another element in the list where the predicate is true

Writing your own iterator with a filter

```
@Override
public T next() {
  int tempPosition = position;
  while (tempPosition < list.size()) {
    T nextElement = list.get(tempPosition);
    if (predicate.test(nextElement)) {
      position=tempPosition+1;
      return nextElement;
    }
    else {
      tempPosition++;
    }
  }
  return null;
}

@Override
public void remove() {
  throw new UnsupportedOperationException();
}
}
```

Find the next element in the list where the predicate is true

Not supported

Using your own filter iterator

```
public class ProductCollection {
  private List<Product> products = new ArrayList<>();

  public void addProduct(Product product){
    products.add(product);
  }

  public Iterator<Product> reverseIterator(){
    return new ReverseIterator<Product>(products);
  }

  public Iterator<Product> filterIterator(Predicate<Product> predicate){
    return new FilterIterator<Product>(products, predicate);
  }
}
```

Factory method creates the iterator

```
public class Product {
  private String number;
  private String name;
  private double price;
  private boolean available;
  ...
}
```

Using your own filter iterator

```java
public class Application {

  public static void main(String[] args) {
    ProductCollection productCollection = new ProductCollection();
    productCollection.addProduct(new Product("A234", "Iphone 10", 850.0, true));
    productCollection.addProduct(new Product("A235", "Iphone 11", 1050.0, false));
    productCollection.addProduct(new Product("A236", "Iphone 9", 650.0, true));
    productCollection.addProduct(new Product("A238", "Iphone 8", 425.0, true));

    System.out.println("Available products:");
    Predicate<Product> availablepredicate = p -> p.isAvailable();
    Iterator<Product> filterIterator = productCollection.filterIterator(availablepredicate);
    while (filterIterator.hasNext()) {
      System.out.println(filterIterator.next());
    }
  }
}
```

```
Available products:
Product [number=A234, name=Iphone 10, price=850.0, available=true]
Product [number=A236, name=Iphone 9, price=650.0, available=true]
Product [number=A238, name=Iphone 8, price=425.0, available=true]
```

Using your own filter iterator

```java
public class Application {

  public static void main(String[] args) {
    ProductCollection productCollection = new ProductCollection();
    productCollection.addProduct(new Product("A234", "Iphone 10", 850.0, true));
    productCollection.addProduct(new Product("A235", "Iphone 11", 1050.0, false));
    productCollection.addProduct(new Product("A236", "Iphone 9", 650.0, true));
    productCollection.addProduct(new Product("A238", "Iphone 8", 425.0, true));

    System.out.println("Products with price > 800:");
    Predicate<Product> pricepredicate = p -> p.getPrice() > 800;
    Iterator<Product> filterIterator = productCollection.filterIterator(pricepredicate);
    while (filterIterator.hasNext()) {
      System.out.println(filterIterator.next());
    }
  }
}
```

```
Products with price > 800:
Product [number=A234, name=Iphone 10, price=850.0, available=true]
Product [number=A235, name=Iphone 11, price=1050.0, available=false]
```

Using your own filter iterator

```
public class Application {

  public static void main(String[] args) {
    ProductCollection productCollection = new ProductCollection();
    productCollection.addProduct(new Product("A234", "Iphone 10", 850.0, true));
    productCollection.addProduct(new Product("A235", "Iphone 11", 1050.0, false));
    productCollection.addProduct(new Product("A236", "Iphone 9", 650.0, true));
    productCollection.addProduct(new Product("A238", "Iphone 8", 425.0, true));

    System.out.println("Available products with price > 800:");
    Predicate<Product> availablepricepredicate = p -> p.getPrice() > 800 && p.isAvailable();
    filterIterator = productCollection.filterIterator(availablepricepredicate);
    while (filterIterator.hasNext()) {
      System.out.println(filterIterator.next());
    }
  }
}
```

```
Available products with price > 800:
Product [number=A234, name=Iphone 10, price=850.0, available=true]
```

Streams

```java
public class ApplicationFilter {

  public static void main(String[] args) {
    List<Product> products = new ArrayList<>();
    products.add(new Product("A234", "Iphone 10", 850.0, true));
    products.add(new Product("A235", "Iphone 11", 1050.0, false));
    products.add(new Product("A236", "Iphone 9", 650.0, true));
    products.add(new Product("A238", "Iphone 8", 425.0, true));

    System.out.println("Available products:");
    List<Product> availableProducts = products.stream()
      .filter(p -> p.isAvailable())
      .collect(Collectors.toList());
    availableProducts.forEach(p -> System.out.println(p));

    System.out.println("Products with price > 800:");
    List<Product> expensiveProducts = products.stream()
      .filter(p -> p.getPrice() > 800)
      .collect(Collectors.toList());
    expensiveProducts.forEach(p -> System.out.println(p));

    System.out.println("Available products with price > 800:");
    List<Product> availableExpensiveProducts = products.stream()
      .filter(p ->  p.isAvailable())
      .filter(p -> p.getPrice() > 800)
      .collect(Collectors.toList());

    availableExpensiveProducts.forEach(p -> System.out.println(p));
  }
}
```

Main point

- The iterator pattern separates the iteration functionality from the collection so that the client is unaware of the structure of the collection.
- When one grows in consciousness, one spontaneously starts to live in harmony with all elements in creation without knowing all the details.

Connecting the parts of knowledge with the wholeness of knowledge

1. The composite pattern creates a tree structure of composites and leaves, and the client treats both elements uniformly.
2. An iterator provides a uniform interface to iterate over a collection of elements .
3. **Transcendental consciousness** is the field at the basis of all creation.
4. **Wholeness moving within itself:** In unity consciousness one realizes that the whole creation is an expression of ones own Self.