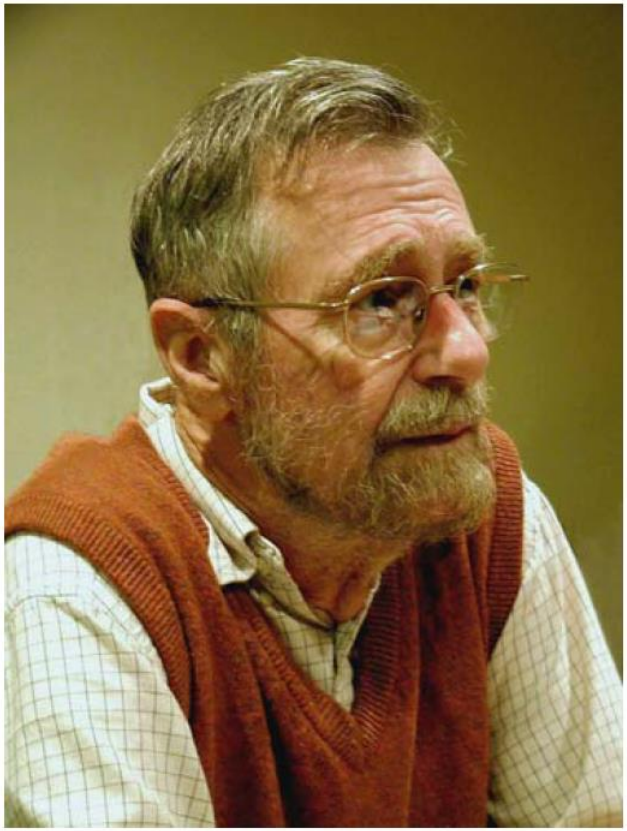
Department of Computer Science
University of Bristol

COMSM0086 – Object-Oriented Programming

COLLECTIONS & STRATEGY

Simon Lock | simon.lock@bris.ac.uk
Sion Hanunna | sh1670@bristol.ac.uk



"…simplicity and elegance are unpopular because they require hard work and discipline to be achieved, and education to be appreciated..."

--- *E. Dijkstra*

"All problems in computer science can be solved by another level of indirection"

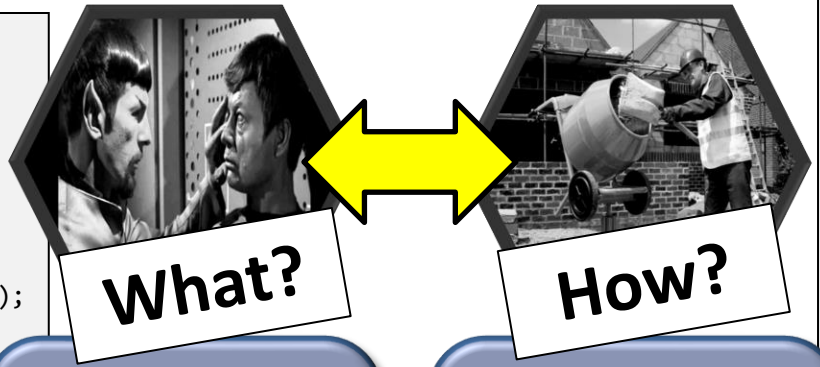
--- *Butler Lampson*

Interfaces
vs.
Concrete Implementations

Interfaces vs. Implementations

- the role of an **`interface`** (e.g. a **`Set`**) is to provide a contract
- any particular concrete implementation (e.g. **`ArraySet`**) has to fullfill it
- an **`interface`** does not force a particular way of realising this contract

```
interface Set<X> {

  public void insert(X x);

  public void delete(X x);

  public void empty();

  public boolean contains(X x);

  public int size();

}
```



interface Set provides a representation-independent contract, which all concrete implementations have to realise

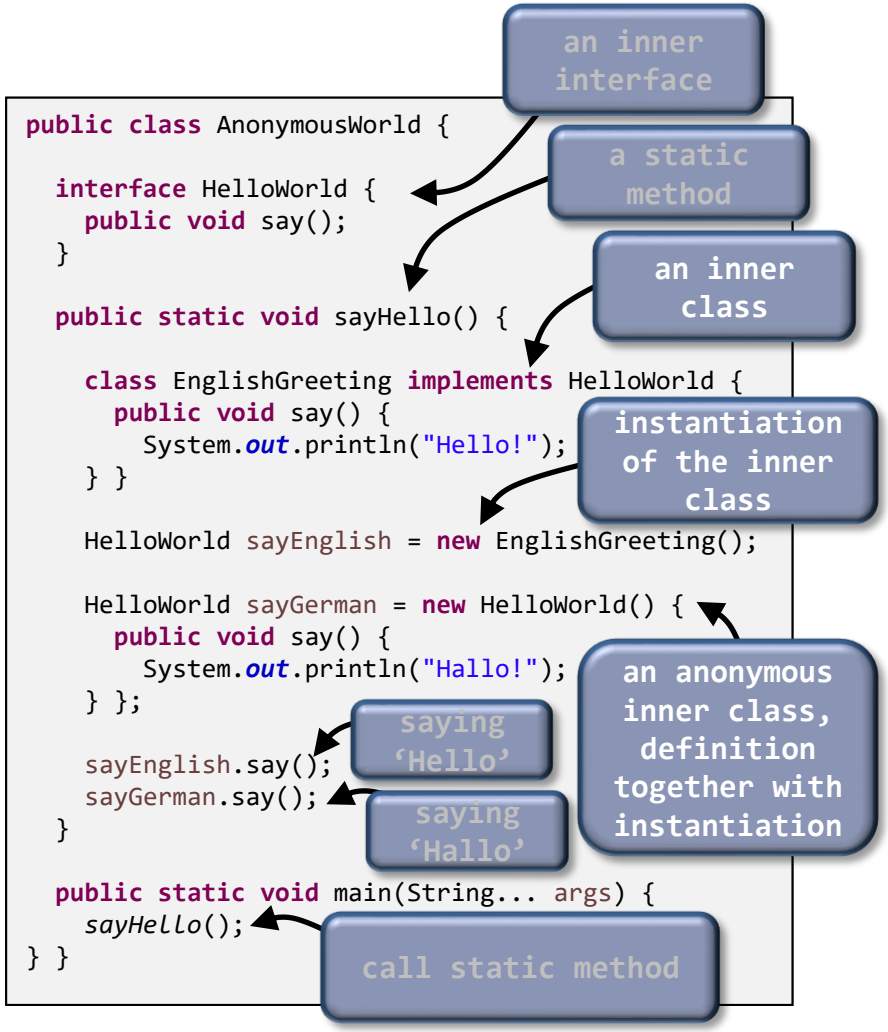
ArraySet implements all methods demanded by the interface Set and specifies a particular, concrete representation of all state and behaviour required

```
class ArraySet<X> implements Set<X> {
  protected X[] values;
  protected int size;
  private final int N = 100;

  public ArraySet() {
    values = (X[]) new Object[N];
    size = 0; }

  @Override
  public void insert(X x) {
    assert(size<100);
    assert(!contains(x));
    values[size] = x;
    size = size + 1; }

  @Override
  public void delete(X x) {
    assert(contains(x));
    for (int i=0; i < size; i = i+1) {
      if (values[i].equals(x)) {
        values[i] = values[size-1];
        size = size - 1;
        break;
  } } }

  @Override
  public boolean contains(X x) {
    boolean contains = false;
    for (X value : values) {
      if (value.equals(x)) {
        contains = true;
        break;
    } }
    return contains; }

  @Override
  public int size() {
    return size; }

  @Override
  public void empty() {
    size = 0;
} }
```

Inner Classes

Inner Classes and Anonymous Classes

- **inner classes** (or inner interfaces too) are defined within another class (the outer class)
- **anonymous (inner) classes** are defined and instantiated in a single expression using **new**, where the anonymous class definition itself is actually an expression
- it can be included as part of a larger expression, such as a method call
- inner classes are often local helper classes, whilst anonymous classes are often use-once classes without an explicit handle to the code that defines it



Iterators

This thing …

```
for(String s : strings){
  System.out.println (s);
}
```

The Concept of Iterators

- various object structures hold elements: e.g. sets, arrays, lists, trees (e.g. **ArraySet** on left)
- we often want to be able to iterate over **all** the elements **independent** of the structure
- Java has an **Iterator** interface to do this (see the JavaDocs for all details)
- classes need to implement **Iterable** to be iterated over using the **:** notation, the interface demands to be able to get hold of an **Iterator** object to drive it

```
class ArraySet<X> implements Set<X> {
  protected X[] values;
  protected int size;
  private final int N = 100;

  public ArraySet() {
    values = (X[]) new Object[N];
    size = 0; }

  @Override
  public void insert(X x) {
    ...
    values[size] = x;
    size = size + 1; }

  @Override
  public void delete(X x) {
    assert(contains(x));
    for (int i=0; i < size; i = i+1) {
      if (values[i].equals(x)) {
        values[i] = values[size-1];
        size = size - 1;
        break;
  } } }

  @Override
  public boolean contains(X x) {
    boolean contains = false;
    for (X value : values) {
      if (value.equals(x)) {
        contains = true;
        break;
    } }
    return contains; }

  @Override
  public int size() {
    return size; }

  @Override
  public void empty() {
    size = 0;
} }
```

Can we make our ArraySet iterable?

Iterable promises to provide an Iterator

```
interface Iterable<E> {
  public Iterator<E> iterator();
...} // shipped with Java
```

```
interface Iterator<E> {
  public boolean hasNext();
  public E next();
...} // shipped with Java
```

```
interface Set<X> {
  public void insert(X x);
  public void delete(X x);
  public void empty();
  public boolean contains(X x);
  public int size();
}
```

an Iterator provides all methods needed to step through all elements of a collection

simple for-loop to iterate through the set

```
class IteratorWorld {
  public static void main (
    String[] args) {
    int sum = 0;
    ...
    IterableArraySet<Integer> set =
      new IterableArraySet<>();
    set.insert(1);
    set.insert(2);
    ...
    for (Integer i : set) {
      sum += i.intValue();
      System.out.println(i);
    }
    System.out.println(sum);
} }
```

interface to implement

Iterator defined as anonymous class

extending the ArraySet to add functionality to iterate over

```
import java.lang.Iterable;
import java.util.Iterator;

class IterableArraySet<X>
      extends ArraySet<X>
      implements
      Iterable<X> {
  @Override
  public Iterator<X> iterator() {
    return new Iterator<X>() {
      private int index = 0;
      public X next () {
        X x = values[index];
        index = index + 1;
        return x;
      }
      public boolean hasNext() {
        return (index < size);
  } };}
}
```

The **Iterator** Pattern *in Detail*

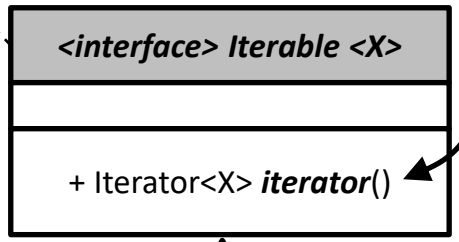
SYNOPSIS | UML | code | comments

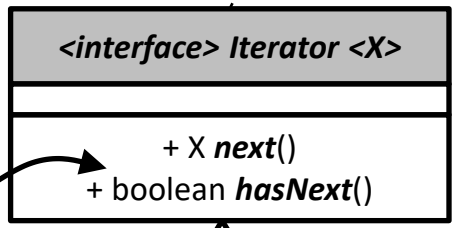
```
interface Iterable<X> {
  public Iterator<X> iterator();
} // shipped with Java
```

collection becomes 'Iterable' once an Iterator for it can be retrieved

```
interface Iterator<X> {
  public boolean hasNext();
  public X next();
} // shipped with Java
```

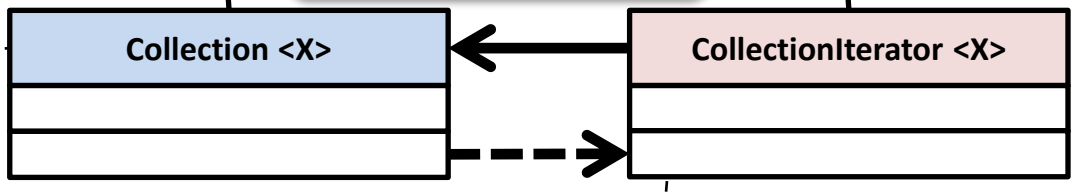
```
interface Set<X> {
  public void insert(X x);
  public void delete(X x);
  public void empty();
  public boolean contains(X x);
  public int size();
}
```






```
class ArraySet<X>
  implements Set<X> {
  protected X[] values;
  protected int size;
  ...
}
```

Iterator has to know how to step through all elements

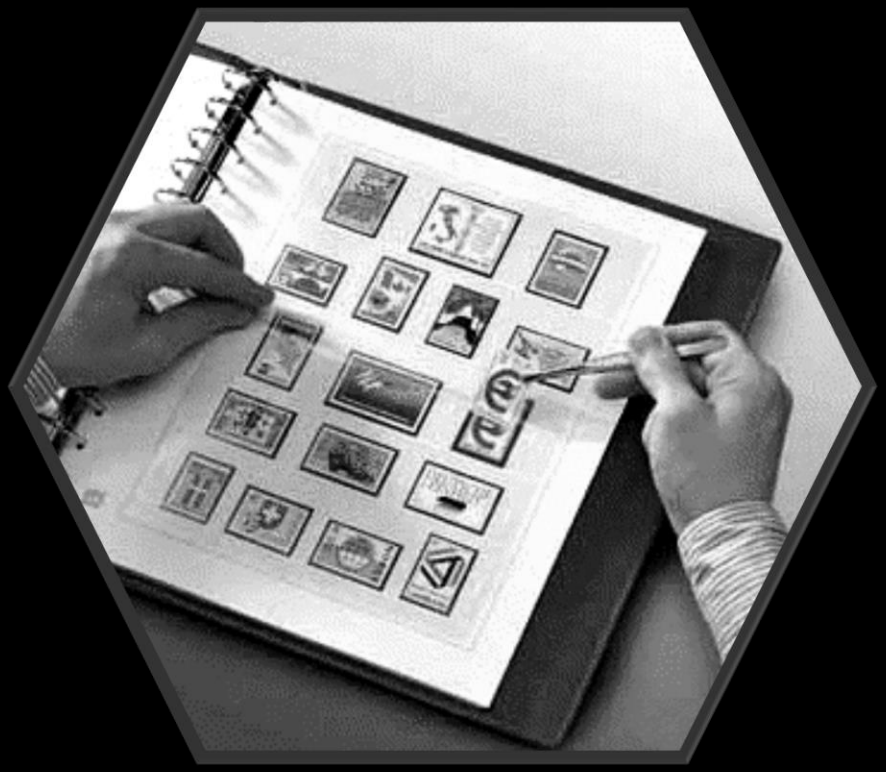



```
class IterableArraySet<X>
        extends ArraySet<X>
        implements
        Iterable<X> {
  @Override
  public Iterator<X> iterator() {
    return new Iterator<X>() {
      private int index = 0;
      public X next () {
        X x = values[index];
        index = index + 1;
        return x;
      }
      public boolean hasNext() {
        return (index < size);
  } };}
}
```

The Iterator pattern is used to provide a standard interface for traversing a collection of items in an aggregate object without the need to understand its underlying structure. [GoF]

```
class IteratorWorld {
    ...
    IterableArraySet<Integer> set =
      new IterableArraySet<>();
    set.insert(1);
    set.insert(2);
    sum = 0;
    for (Integer i : set) {
      sum += i.intValue();
      System.out.println(i);
    }
    System.out.println(sum);
} }
```

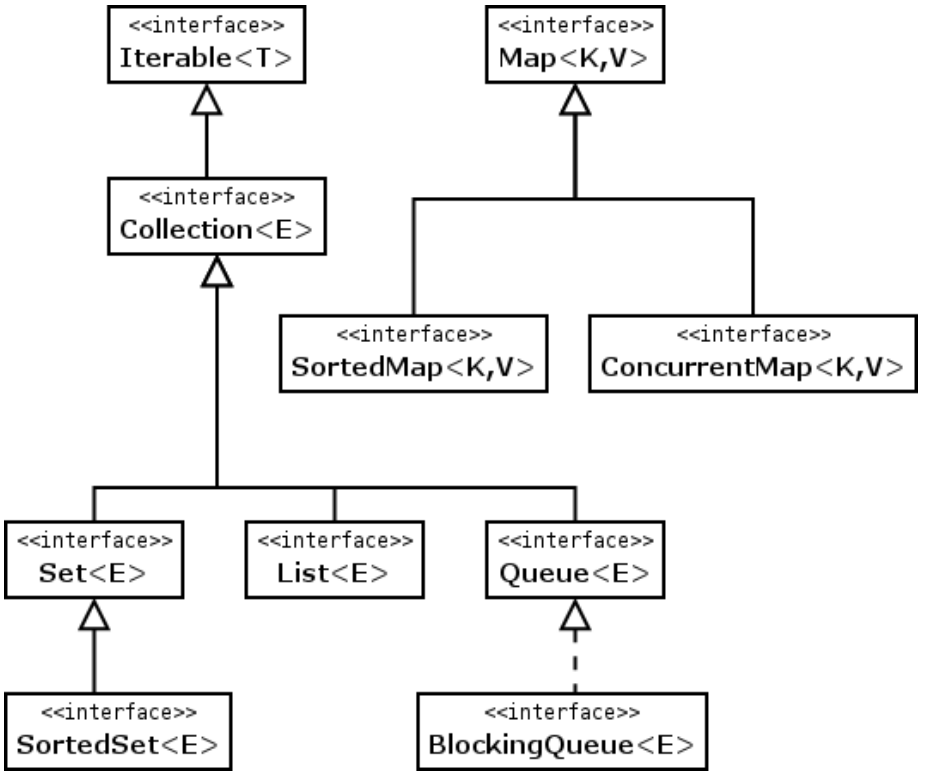
Collections



Java Collections

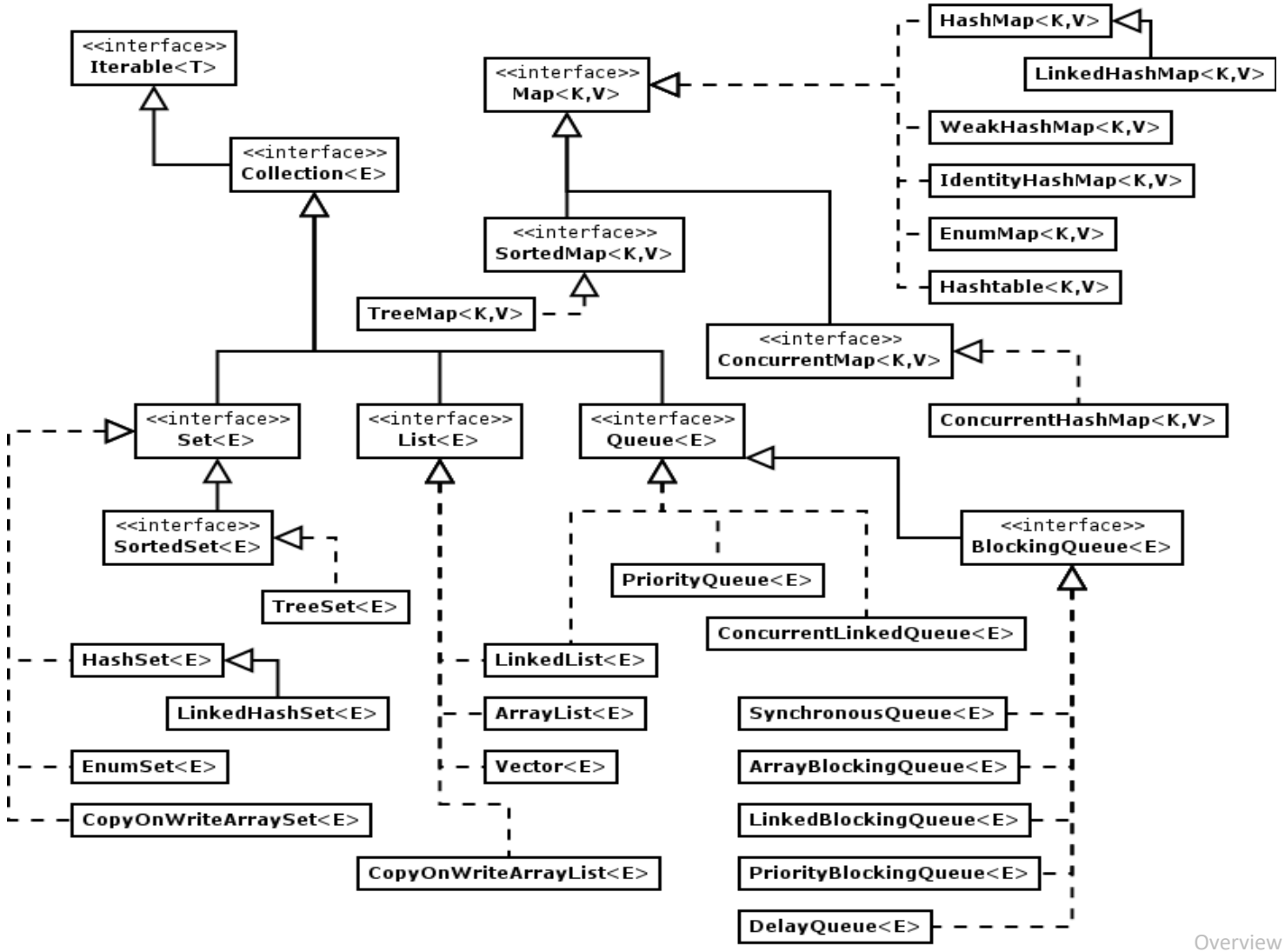
- a **`Collection`** is a container that groups multiple elements into a single unit
- the Java **`Collection`** framework (JCF) is one of the most important ones in all of Java's libraries, providing high performance implementations
- it uses generics to be flexible w.r.t. element types contained, and it is also **polymorphically structured**, so the same methods work on different collections

Interfaces of the Collections Framework



Overview by Ray Toal

JCF Summary


<<interface>>
Iterable<T>
<<interface>>
Collection<E>
<<interface>>
Map<K,V>
HashMap<K,V>
LinkedHashMap<K,V>
WeakHashMap<K,V>
IdentityHashMap<K,V>
EnumMap<K,V>
Hashtable<K,V>
<<interface>>
SortedMap<K,V>
TreeMap<K,V>
<<interface>>
ConcurrentMap<K,V>
ConcurrentHashMap<K,V>
<<interface>>
Set<E>
<<interface>>
List<E>
<<interface>>
Queue<E>
<<interface>>
SortedSet<E>
TreeSet<E>
PriorityQueue<E>
<<interface>>
BlockingQueue<E>
ConcurrentLinkedQueue<E>
HashSet<E>
LinkedHashSet<E>
EnumSet<E>
CopyOnWriteArraySet<E>
LinkedList<E>
ArrayList<E>
Vector<E>
CopyOnWriteArrayList<E>
SynchronousQueue<E>
ArrayBlockingQueue<E>
LinkedBlockingQueue<E>
PriorityBlockingQueue<E>
DelayQueue<E>
Overview by Ray Toal

Example Usage: The List<E> Interface

```
import java.util.*;

class ListWorld {
  static void printList(List<Robot> list) {
    System.out.print("List is:");
    for (Robot robot : list) {
      System.out.print(robot.name+',');
    }
    System.out.println("");
  }

  public static void main(String args[]) {
    List<Robot> list = new ArrayList<>();
    Robot c3po = new Robot("C3PO");
    list.add(c3po);
    list.add(new CarrierRobot());
    printList(list);
    list.add(1, new Robot("C4PO"));
    printList(list);
    Robot removed = list.remove(2);
    System.out.println("Removed:"+
      removed.name);
    printList(list);
    System.out.println("C3PO in list?:"+
      list.contains(c3po));
    list.addAll(0,list);
    printList(list);
} }
```

import of the right package

simple for-loop to iterate through the set

construct list object (note type inference)

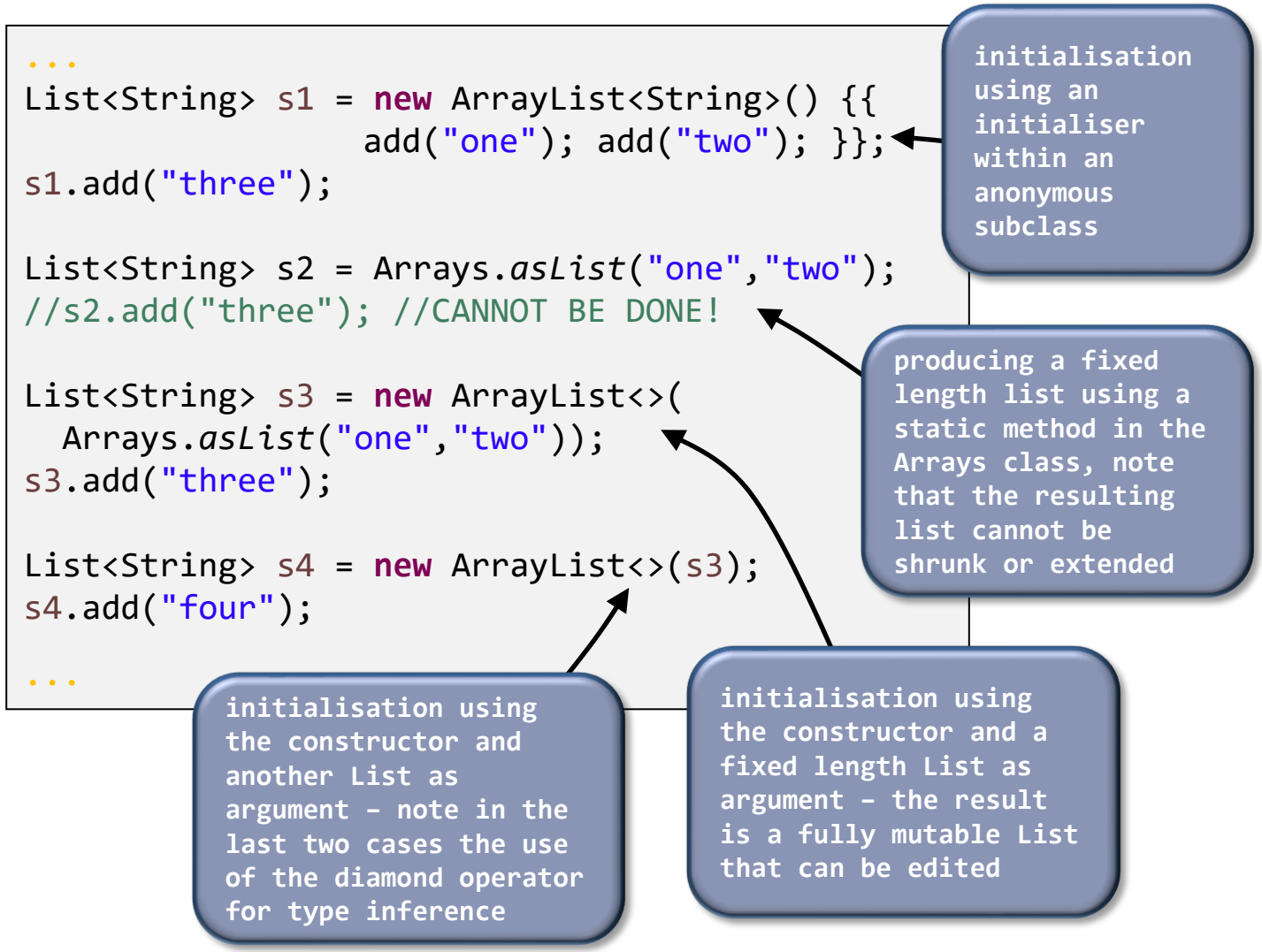
adding elements

element removal

«interface» List<E>
+ boolean **add**(int index, E)
+ boolean **addAll**(int index, Collection<E>)
+ void **clear**()
+ boolean **contains**(Object o)
+ boolean **containsAll**(Collection c)
+ E **get**(int index)
+ int **indexOf**(Object)
+ int **lastIndexOf**(Object)
+ E **remove**(int index)
+ E **set**(int index, E)
+ Iterator<E> **iterator**()
+ ListIterator<E> **listIterator**()
+ List<E> **subList**(int fromIndex, int toIndex)
+ int **size**()
+ boolean **isEmpty**()

```
List is:C3PO,Standard Model,
List is:C3PO,C4PO,Standard Model,
Removed:Standard Model
List is:C3PO,C4PO,
C3PO in list?:true
List is:C3PO,C4PO,C3PO,C4PO,
```

Some List Initialisation Options



Comparing

Comparing Objects

- objects often have a natural ordering: a way in which it makes sense for them to be compared
- there are some obvious examples, such as numbers: 3 < 10
- objects in a class usually need to be compared by some natural ordering: we might consider the weight of an animal

abstract class instead of interface allows to define standard behaviour

```
abstract class Animal {
  protected int weight;
  abstract void makeNoise();
  int getWeight() { return weight; }
  boolean weighsLessThan(Animal that) {
    return (this.getWeight() < that.getWeight());
} }
```

```
class Elephant extends Animal {
  Elephant() {
    weight = 1000000;
  }
  void makeNoise() {
    System.out.println("Trumpet..");
} }
```

```
class Ant extends Animal {
  Ant () {
    weight = 5;
  }
  void makeNoise() {
    System.out.println("Khllkhll..");
} }
```

<

an ant called Dec

an elephant called Nellie

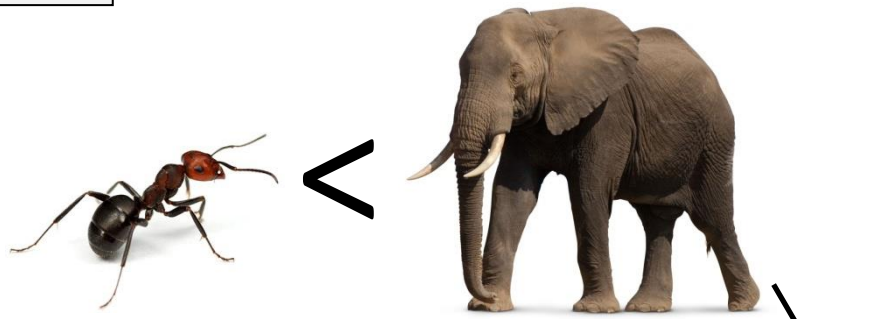
slide content extended from N Wu

A Basic Comparison

```
abstract class Animal {
  protected int weight;
  abstract void makeNoise();
  int getWeight() { return weight; }
  boolean weighsLessThan(Animal that) {
    return (this.getWeight() < that.getWeight());
} }
```

```
class Elephant extends Animal {
  Elephant() {
    weight = 1000000;
  }
  void makeNoise() {
    System.out.println("Trumpet..");
} }
```

```
class Ant extends Animal {
  Ant () {
    weight = 5;
  }
  void makeNoise() {
    System.out.println("Khllkhll..");
} }
```



comparison now possible for ALL Animal objects

```
class CompareWorld {
  public static void main (String[] args) {
  Animal dec = new Ant();
  Animal nellie = new Elephant();
  System.out.println(dec.weighsLessThan(nellie));
} }
```

slide content extended from N Wu

Considerations towards Comparators

- in theory, we could use **`weighsLessThan()`** to sort a list of animals, but it's not completely satisfactory in the general case:
- it only works for the **`Animal`** class and its children - other classes need comparisons too!
- **`weighsLessThan()`** returns a **`boolean`**: we might want to provide some further information on, for instance, by how much they differ
- thus, a richer method **`compareWeightTo()`** could be used to return some measure of difference between the two objects

```
abstract class Animal {
  protected int weight;
  abstract void makeNoise();
  int getWeight() {
    return weight;
  }
  boolean weighsLessThan(Animal that) {
    return (this.getWeight() < that.getWeight());
  }
  public int compareWeightTo(Animal that) {
    return (this.getWeight() - that.getWeight());
} }
```

```
class CompareWorld {
  public static void main (String[] args) {
  Animal dec = new Ant();
  Animal nellie = new Elephant();
  System.out.println(dec.weighsLessThan(nellie));
  System.out.println(dec.compareWeightTo(nellie));
} }
```

this comparison now provides information on the weight difference

`.compareWeightTo(`

`) = -999995`

slide content extended from N Wu

An Interface for Comparing

- we realise that we need a way of unifying different comparison methods – we could add a method **`compareTo()`** to the **`Object`** class, couldn't we … why is this a bad idea?
- instead of imposing a single **`compareTo()`** method that works on everything, we should use a generic **`interface`** that guarantees that comparisons can only be made between related objects:

```
public interface Comparable<T> { //Java provides this interface!
  int compareTo(T that);
}
```

- this allows us to produce various different classes as implementations of this interface...

```
abstract class Animal implements Comparable<Animal> {
  protected int weight;
  abstract void makeNoise();
  int getWeight() { return weight; }
  public int compareTo(Animal that) {
    return (this.getWeight() - that.getWeight());
} }
```

implementation of the interface declared for all animals

implementation of the compareTo() method is provided here for all animals

Sorting using the Comparable Interface

the class Collections provides static helper methods such as 'sort' for various collections – note the specific type parameter specification

```
...
public class Collections ...

public static <T extends Comparable<? super T>> void sort(List<T> list);
// T can implement Comparable<? super T>, not just Comparable<T>. For example:
//class TranslationRobot extends Robot implements Comparable <Robot> //sorted by Robot properties
```

```
public interface Comparable<T> {
  int compareTo(T that);
}
```

```
abstract class Animal implements Comparable<Animal> {
  protected int weight;
  abstract void makeNoise();
  int getWeight() { return weight; }
  public int compareTo(Animal that) {
    return (this.getWeight() - that.getWeight());
} }
```

once we have implemented the Comparable interface, our elements have a natural ordering

```
class Elephant extends Animal {
  Elephant() {
    weight = 1000000;
  }
  void makeNoise() {
    System.out.println("Trumpet..");
} }
```

```
class Ant extends Animal {
  Ant () {
    weight = 5;
  }
  void makeNoise() {
    System.out.println("Khllkhll..");
} }
```

```
import java.util.*;

class CompareWorld {
  public static void main (String[] args) {
  List<Animal> animals = new ArrayList<Animal>() {
    { add(new Elephant());
      add(new Ant());
  } };
  Collections.sort(animals);
  System.out.println(animals);
} }
```

implementation of the Comparable interface by a class (e.g. Animal) allows us to sort lists of this class in a single line

Naturally Sorted Collections

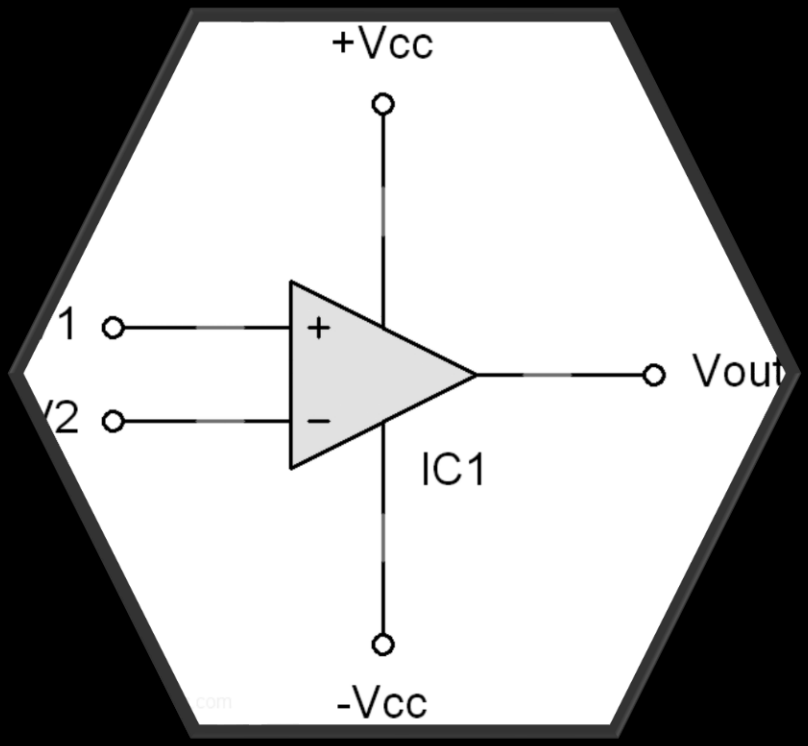
- some collections, such as **`SortedSet`**, store all the elements in their natural ordering anyway
  (remember sorted binary trees – they are a TreeSet in Java)
- since any **`SortedSet`** is also **`Iterable`** the corresponding iterator is also sorted!

```
import java.util.*;

class CompareWorld {
  public static void main (String[] args) {
  SortedSet<Animal> animals = new TreeSet<Animal>() {
    { add(new Elephant());
      add(new Ant());
  } };
  for (Animal a : animals) {
    System.out.println(a);
} } }
```

creating a TreeSet of Animal objects will guarantee that the elements of the set are internally stored in their natural order at the moment of insertion

as a consequence, iterating through the elements of the SortedSet produces a naturally ordered sequence of going through the elements starting with the smallest...

Comparators



Motivation for Comparators

- sometimes the natural ordering is not what you're after:

  - the nodes in a graph might be ordered by their name, but you want them ordered by the number of neighbours

  - animals might be compared by their height instead of their weight
  - there might be no natural ordering!
- sometimes the existing legacy code might be unmodifiable, and without a defined ordering
- in short: sometimes we would like to provide the means to comparing objects after the classes concerned have been finalised with possibly different variants/implementations for specific comparisons
- therefore, instead of forcing a class to be **`Comparable`**, you can instead provide one or many implementations of **`Comparator`**:

```
interface Comparator<X> {
  int compare(X x1, X x2);
}
```

the interface Comparator is provided by the JCF and demands implementers provide a method to compare objects of some common type – note that these objects are now parameters rather than receivers; the code for comparing can therefore live outside the existing classes in a new class that implements Comparator

slide content extended from N Wu

Sorting using Comparators

```
import java.util.Comparator;
public class RobotLegsComparator implements Comparator<Robot> {
  public int compare(Robot robotA, Robot robotB) {
    return (robotA.numLegs - robotB.numLegs);
} }
```

```
import java.util.Comparator;
class RobotPowerComparator implements Comparator<Robot> {
  public int compare(Robot robotA, Robot robotB) {
    return (Math.round(robotA.powerLevel - robotB.powerLevel));
} }
```

```
import java.util.*;

class CompareWorld {
  public static void main (String[] args) {
  List<Robot> robots = new ArrayList<Robot>() {
          { add(new CarrierRobot());
            add(new Robot("C3PO"));
  } };
  Collections.sort(robots, new RobotLegsComparator());
  System.out.println(robots);
  robots.get(0).charge(10);
  Collections.sort(robots, new RobotPowerComparator());
  System.out.println(robots);
} }
```

we can now define various different implementations that all provide different ways of comparing objects of a class, for instance Robots

the simplest way of using comparators for sorting is by provision as the second parameter to the static 'sort' method of Collections

More Ways of using Comparators

```
import java.util.*;

class CompareWorld {
  public static void main (String[] args) {
  List<Robot> robots = new ArrayList<Robot>() {
    { add(new CarrierRobot());
      add(new Robot("C3PO"));
    } };
  robots.get(0).charge(10);
  robots.sort(new RobotPowerComparator());
  System.out.println(robots);
} }
```

DYNAMIC CALL: using a Comparator instance as parameter to the 'sort' method of the instance of List we are using – this utilisation avoids the use of static methods

CONSTRUCTOR PARAMETER: if the attributes/state that underpin the Comparator are not changing after filling of the Collection then one can use a Comparator object as parameter of a constructor of a naturally sorted Collection

```
import java.util.*;

class CompareWorld {
  public static void main (String[] args) {
    SortedSet<Robot> robots =
      new TreeSet<Robot>(new RobotPowerComparator());
    Robot c3po = new Robot("C3PO");
    c3po.charge(10);
    robots.add(new CarrierRobot());
    robots.add(c3po);
    System.out.println(robots);
} }
```

A Design Pattern Emerges

- We have solved a **general problem** here:
  whenever there are multiple ways of solving a problem (like on what to sort), we can hand-over an object (like a comparator) to the solution algorithm (like quicksort) that details which [sorting] strategy to implement.

this is a new design pattern! –
The Strategy Pattern

The Strategy Pattern

Strategy Pattern *in Detail*

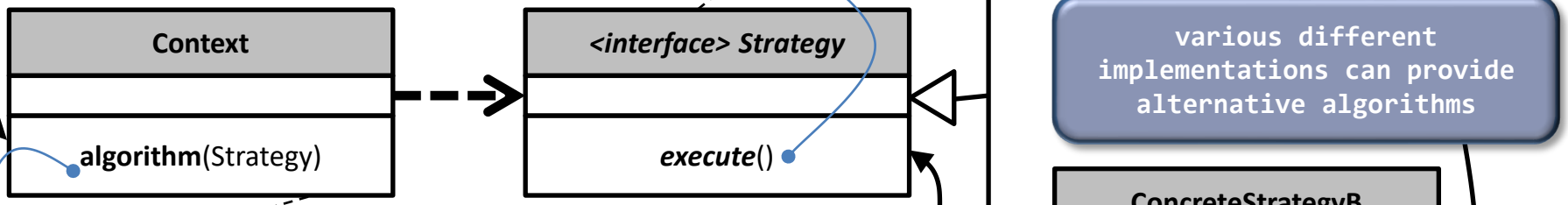
SYNOPSIS | UML | code | comments

The Strategy Pattern defines a set of encapsulated algorithms that can be swapped to carry out a specific behaviour. [GoF]

```
import java.util.Comparator;
public class RobotLegsComparator implements Comparator<Robot> {
  public int compare(Robot robotA, Robot robotB) {
    return (robotA.numLegs - robotB.numLegs);
} }
```

calling the 'algorithm' method with a concrete Strategy object triggers execution – 'algorithm' (sort) uses 'execute' (compare), but does not rely on its specific implementation

```
interface Comparator<X> {
  int compare(X x1, X x2);
} // shipped with Java
```




```
import java.util.*;

class CompareWorld {
  public static void main (String[] args) {
    List<Robot> robots = new ArrayList<Robot>() {
      { add(new CarrierRobot());
        add(new Robot("C3PO"));
      } };
    robots.get(0).charge(10);
    robots.sort(new RobotPowerComparator());
    robots.sort(new RobotLegsComparator());
} }
```

```
import java.util.Comparator;
class RobotPowerComparator implements Comparator<Robot> {
  public int compare(Robot robotA, Robot robotB) {
    return (Math.round(robotA.powerLevel - robotB.powerLevel));
} }
```

To Do

- recap content and check out the unit website
- write, compile, run and understand **ALL** the tiny programs from the lectures so far

→Remember that we currently recommend 7 hours of time to go into this unit per week overall – work in your team of two as often as possible. *Ask yourself: Would you be ready for a theory exam on OO tomorrow? Can you read and write tiny programs using all concepts in any of the lectures so far?*