COM1003 Java Programming

Richard Clayton

Collector's Items

The Java Collections Framework (JCF)

What is a Collection?

A collection is simply an object that is responsible for grouping lots of elements together

- A poker hand (a collection of cards)
- An inbox (a collection of emails)
- A payroll (a collection of employees)

Collections are used to do things with data of the same type

- Store
- Retrieve
- Manipulate
- Communicate

What Java Collection have we already met so far?

Arrays

You need to specify how many elements you will need in advance

- If you don't know, you tend to have to over-allocate - specify a larger size than you actually think you'll need

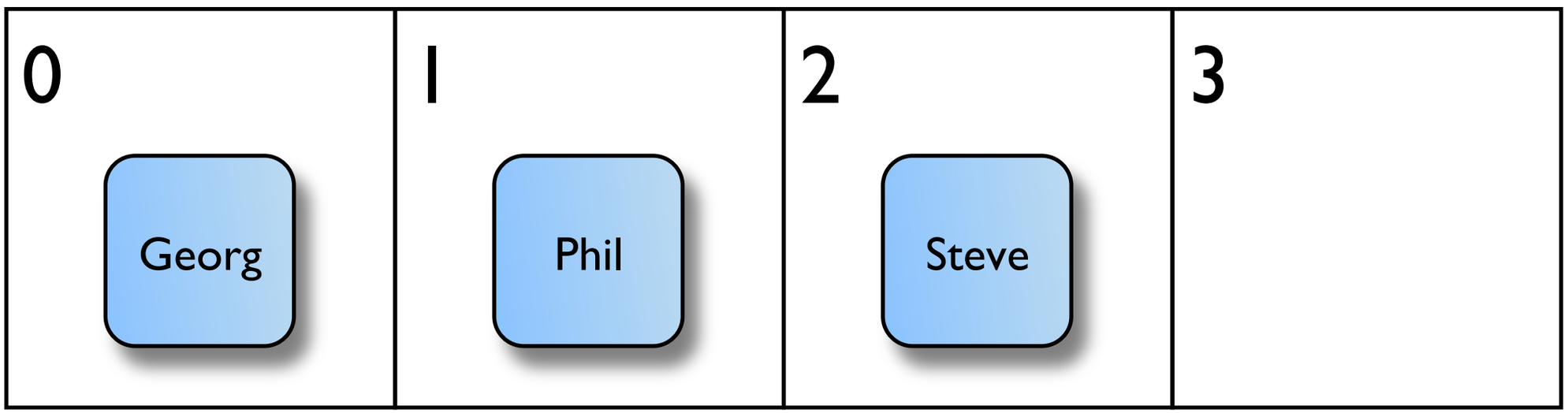
Insertion into the middle of the array means you have to move the later elements up an index

Deletion from the middle of the array means we have to movr later elements down an index

Insertion with arrays

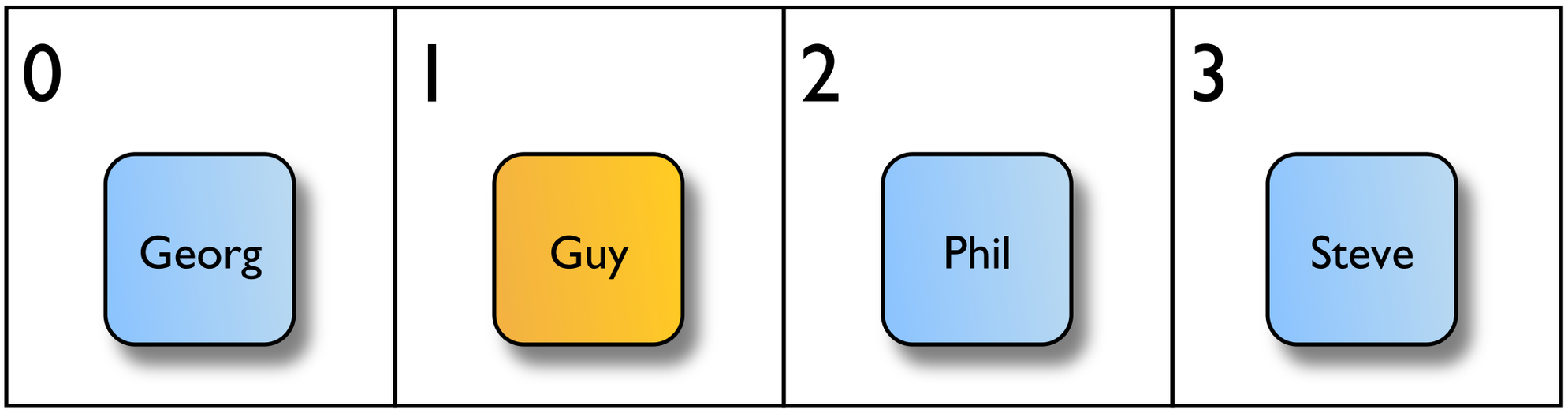
String[4] lecturers = new String[4];

Guy



Deletion with arrays

String[4] lecturers = new String[4];



Lists

The Java Collections Framework contains ***List*** classes that are similar to arrays but are much more flexible.

They automatically grow and shrink in size.

- No need to specify in advance how big the list needs to be

They automatically manage insertions and deletions.

- No need to write code to shunt elements up and down.

ArrayList

```java
import java.util.ArrayList;

// ...

ArrayList<String> lecturers = new ArrayList<String>();

lecturers.add("Georg");
lecturers.add("Phil");
lecturers.add("Steve");

// add Guy at position 1
lecturers.add(1, "Guy");

// remove Guy at position 1
lecturers.remove(1);
```

ArrayList lives in the java.util package. We need to import it before using it, as shown

Construction is like any normal object, except we must specify the class of objects which it will contain. (We'll come back to this later)

No need for shunting elements about - ArrayList moves them up and down for us.

Special for-each loop

The variable containing the next element of the Collection for use in the loop body

The Collection being iterated over

```
for (String lecturer : lecturers) {
    System.out.println(lecturer);
}
```

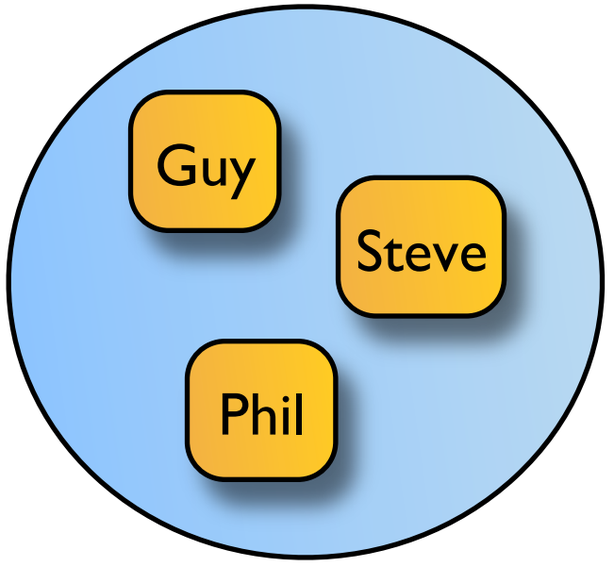
Iterating using indexes is tiresome

Java has an enhanced for loop called the for-each loop that we can use with Collections (including arrays)

Other types of Collection

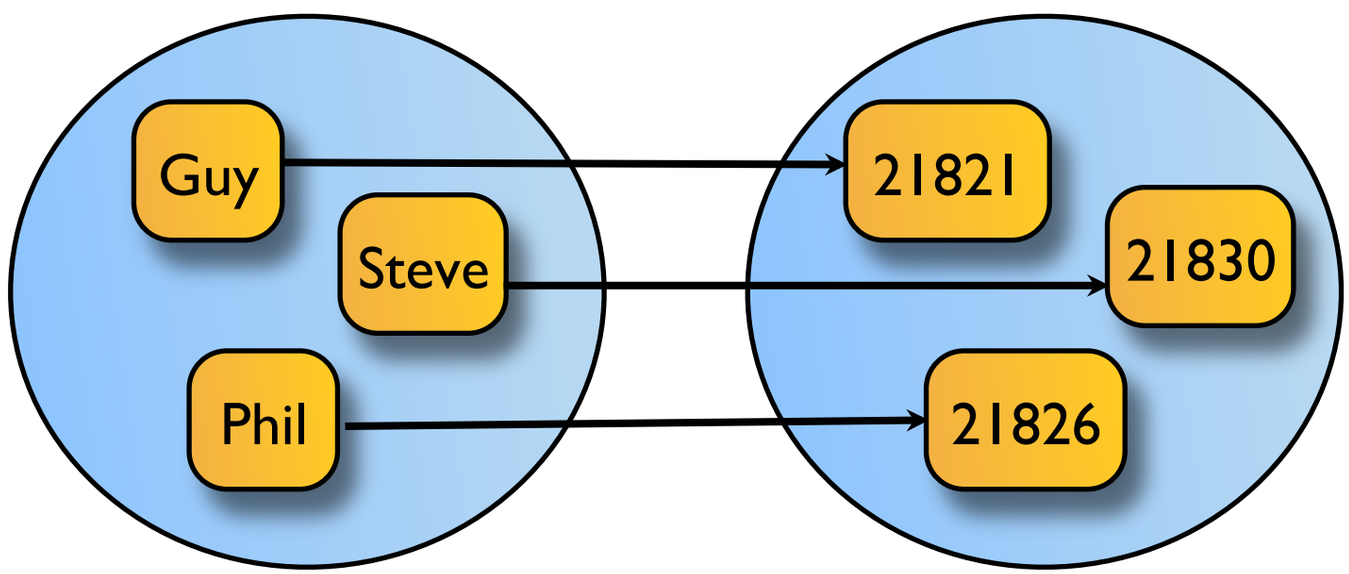
Sets

- Collections of unique items
- Order is not important



Maps

- Let you find or lookup objects in a collection quickly using other objects, called “keys”



What is a Framework?

A framework is a set of interfaces and classes that help us develop programs quickly.

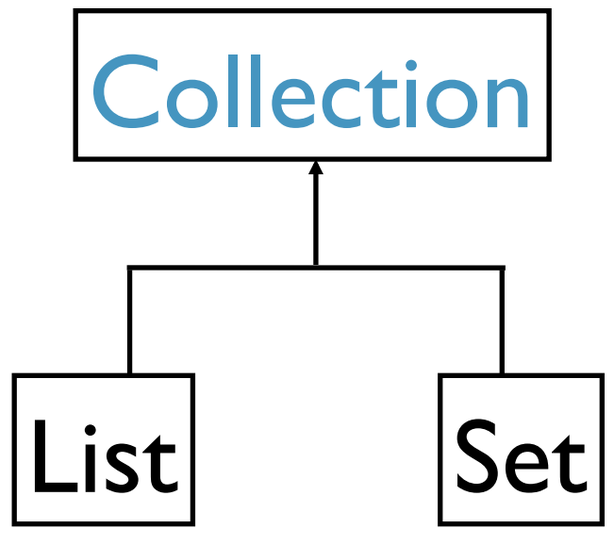
The Java Collections Framework is a framework that

- Provides different types of collection for using in our programs
  - Lists, sets and maps
- Provides common ways of using those collections
  - So that we can change our programs easily if we need to
- Provides methods using common algorithms for common tasks involving collections
  - Sorting and searching

Collections

The Collection interface

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```



General methods to add and remove objects, get the size of the container, check if it's empty.

Lives in java.util

Understanding the syntax

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

"E" refers to the class of objects that will be placed in the collection.

E acts a placeholder because the class of objects that the collection will store is not known until it is instantiated (recall the angle brackets we used with ArrayList).

So errors can be picked up by the compiler rather than appearing as runtime bugs.

This angled bracket notation is a special aspect of Java syntax that is part of Java's "Generics" mechanism.

Understanding the syntax

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The "?" means "any class", i.e. a Collection of any class can be passed to this method

Understanding the syntax

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

"? extends E" means "any class than extends E".

The method can be supplied with a Collection containing objects of a class that extends the class of objects stored in this particular Collection.

The Collection interface

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

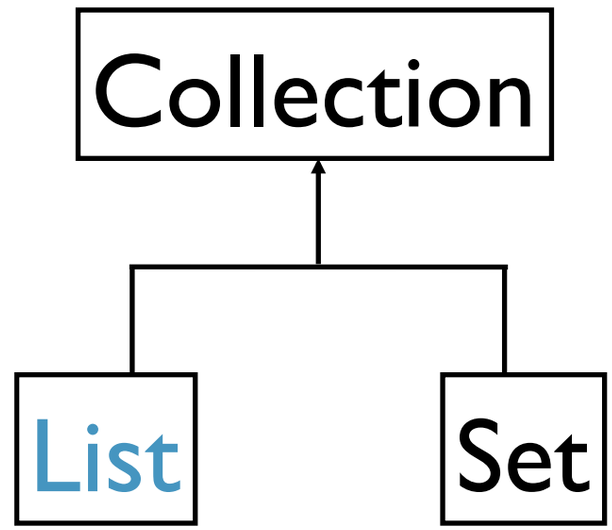
"T" is another placeholder, this time for the class of the array that is passed to the method.

Using this method we can convert the Collection of objects of class E to an array of objects of class T.

Iterators

The List Interface

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    // optional
    E set(int index, E element);
    // optional
    boolean add(E element);
    // optional
    void add(int index, E element);
    // optional
    E remove(int index);
    // optional
    boolean addAll(int index, Collection<? extends E> c);

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```



General methods to add and remove objects from the list, but this time with index parameters.

Lives in java.util

Iterators

ArrayList (as well as TreeSet, HashSet, and LinkedList) implements Collection and returns an Iterator when the iterator method is invoked.

Iterators are objects that can be used to visit the elements of a collection one by one. Here is the Iterator interface they must implement:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Returns true if the iterator can be moved forward

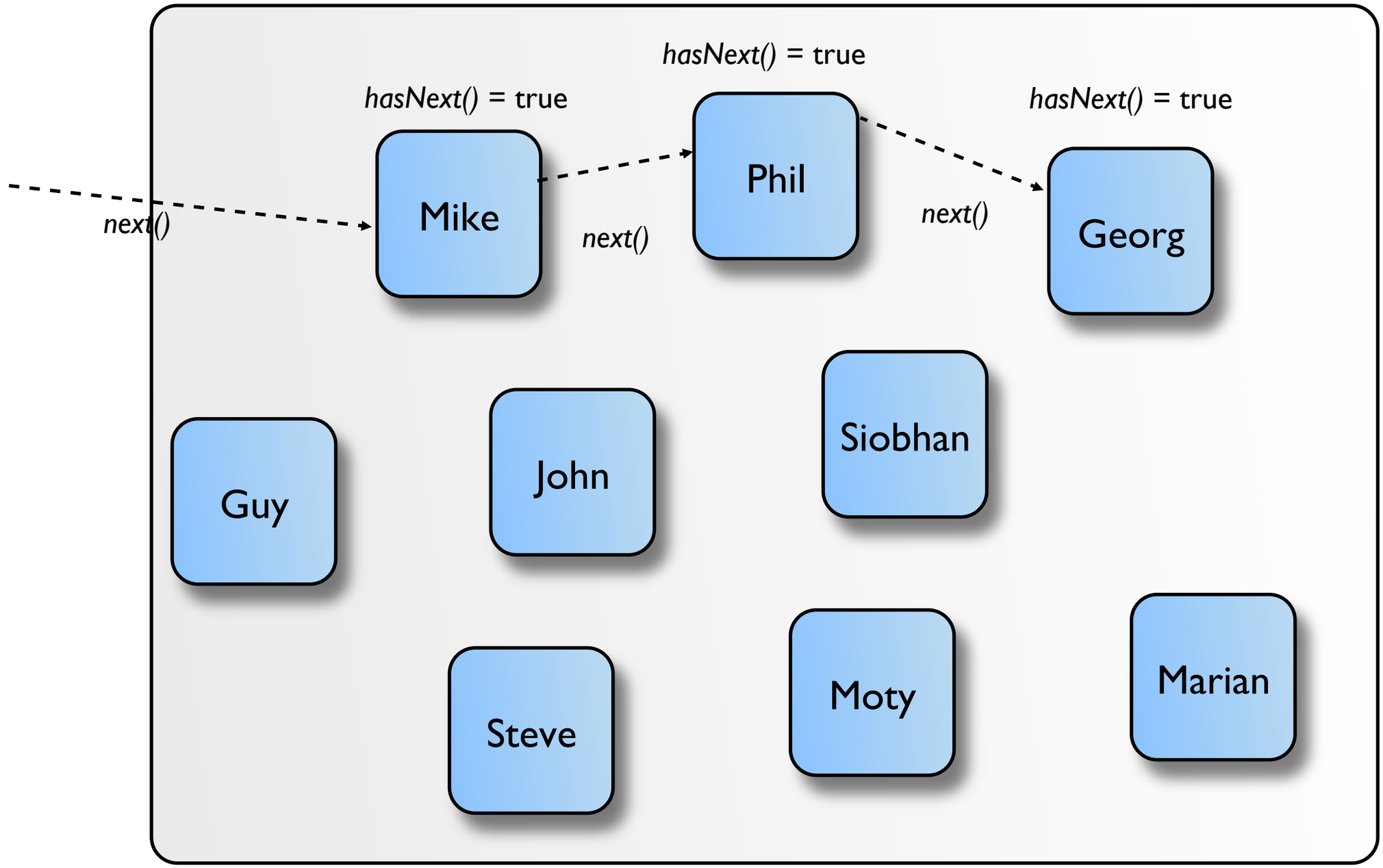
Moves the iterator forward one position, returning the element it passed over

Removes the element just passed over by next()

```
Collection<Lecturer> lecturers = new ArrayList<Lecturer>();

lecturers.add(new Lecturer("Georg Struth"));
lecturers.add(new Lecturer("Moty Katzman"));
lecturers.add(new Lecturer("Guy Brown"));
lecturers.add(new Lecturer("Joab Winkler"));
lecturers.add(new Lecturer("Phil McMinn"));
lecturers.add(new Lecturer("Steve Maddock"));
lecturers.add(new Lecturer("Mike Stannett"));
lecturers.add(new Lecturer("Siobhan North"));
lecturers.add(new Lecturer("Marian Gheorghe"));

Iterator<Lecturer> iterator = lecturers.iterator();

while (iterator.hasNext()) {
    Lecturer lecturer = iterator.next();

    if (lecturer.getName().equals("Joab Winkler")) {
        iterator.remove();
    }

    System.out.println(lecturer.getName());
}
```

This will remove Joab from the lecturers list. Note that remove can only be called after next.

Iterator makes no guarantee on the order in which the elements are returned. Elements may be returned in a potentially arbitrary order.



ListIterator

Of potentially more use for List collections is ListIterator, which is returned by the listIterator of collections implementing the List interface (LinkedList and ArrayList).

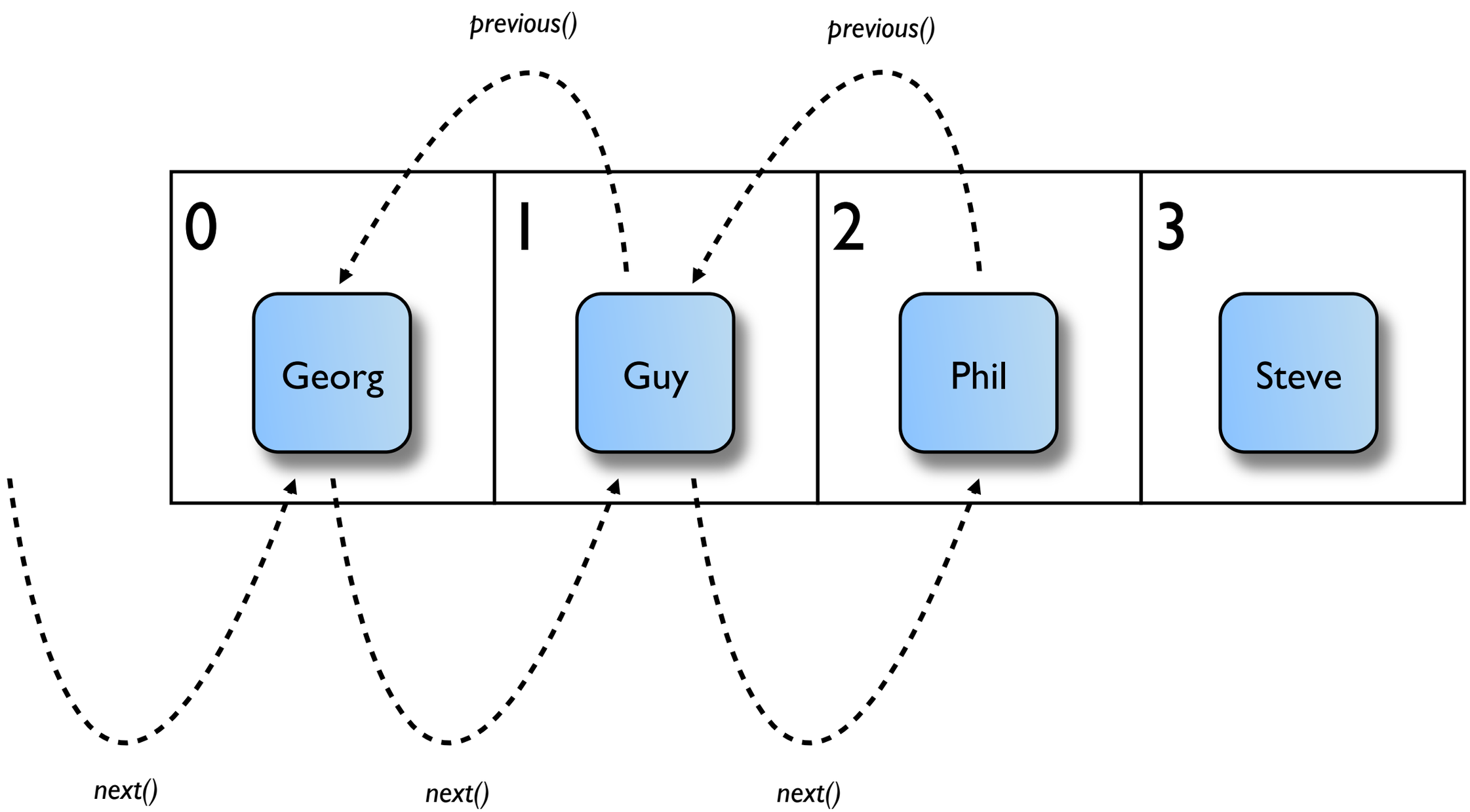
ListIterator returns elements in list order.

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove();
    void set(E e);
    void add(E e);
}
```

Reverse traversal is supported

Can explicitly *add* elements to the container or overwrite existing ones with *set*

```
List<Lecturer> lecturers = new ArrayList<Lecturer>();

lecturers.add(new Lecturer("Georg Struth"));
lecturers.add(new Lecturer("Moty Katzman"));
lecturers.add(new Lecturer("Guy Brown"));
lecturers.add(new Lecturer("Joab Winkler"));
lecturers.add(new Lecturer("Phil McMinn"));
lecturers.add(new Lecturer("Steve Maddock"));
lecturers.add(new Lecturer("Mike Stannett"));
lecturers.add(new Lecturer("Siobhan North"));
lecturers.add(new Lecturer("Marian Gheorghe"));

ListIterator<Lecturer> listIterator = lecturers.listIterator();

while (listIterator.hasNext()) {
    int index = listIterator.nextIndex();
    Lecturer lecturer = listIterator.next();

    System.out.println(index+") "+lecturer.getName());

    if (lecturer.getName().equals("Phil McMinn")) {
        listIterator.add(new Lecturer("Mike Holcombe"));
        // go back one so that the added element is printed out
        // in the next iteration
        listIterator.previous();
    }
}
```


previous()
previous()
0
1
2
3
Georg
Guy
Phil
Steve
next()
next()
next()
hasNext() = true
hasPrevious() = false
hasNext() = true
hasPrevious() = true
hasNext() = true
hasPrevious() = true

For-each loop revisited

Recall the for-each loop:

```
for (Lecturer l : lecturers) {
    // do some stuff
}
```

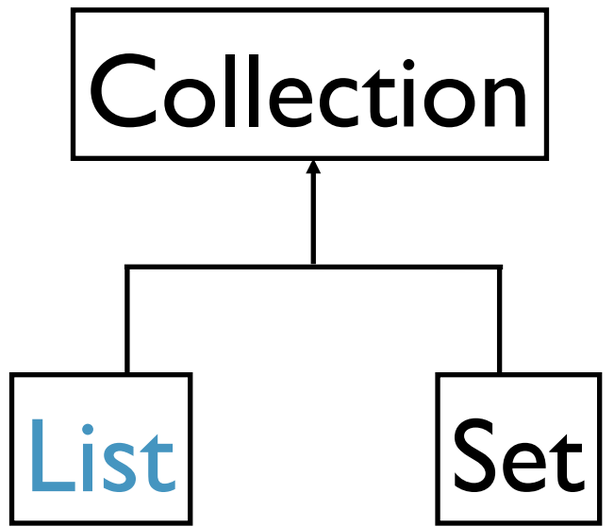
This is actually Java using the Collection's iterator.
It is (the less verbose) equivalent to writing:

```
for (Iterator<Lecturer> iterator = lecturers.iterator(); iterator.hasNext(); ) {
    Lecturer l = iterator.next();
    // do some stuff
}
```

Lists

Properties of a List

- Its elements are ordered
- Its elements may be duplicated

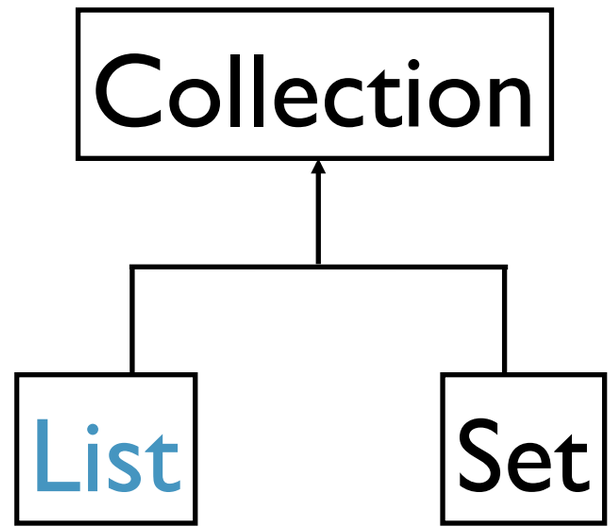


List is implemented by

- ArrayList (already met)
- LinkedList (lives in java.util)

The List Interface

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    // optional
    E set(int index, E element);
    // optional
    boolean add(E element);
    // optional
    void add(int index, E element);
    // optional
    E remove(int index);
    // optional
    boolean addAll(int index, Collection<? extends E> c);

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```



General methods to add and remove objects from the list, but this time with index parameters.

Lives in java.util

ArrayList

ArrayList implements the List interface (http://www.docs.oracle.com/javase/8/docs/api)

ArrayList uses an internal array.

ArrayList's capacity grows automatically

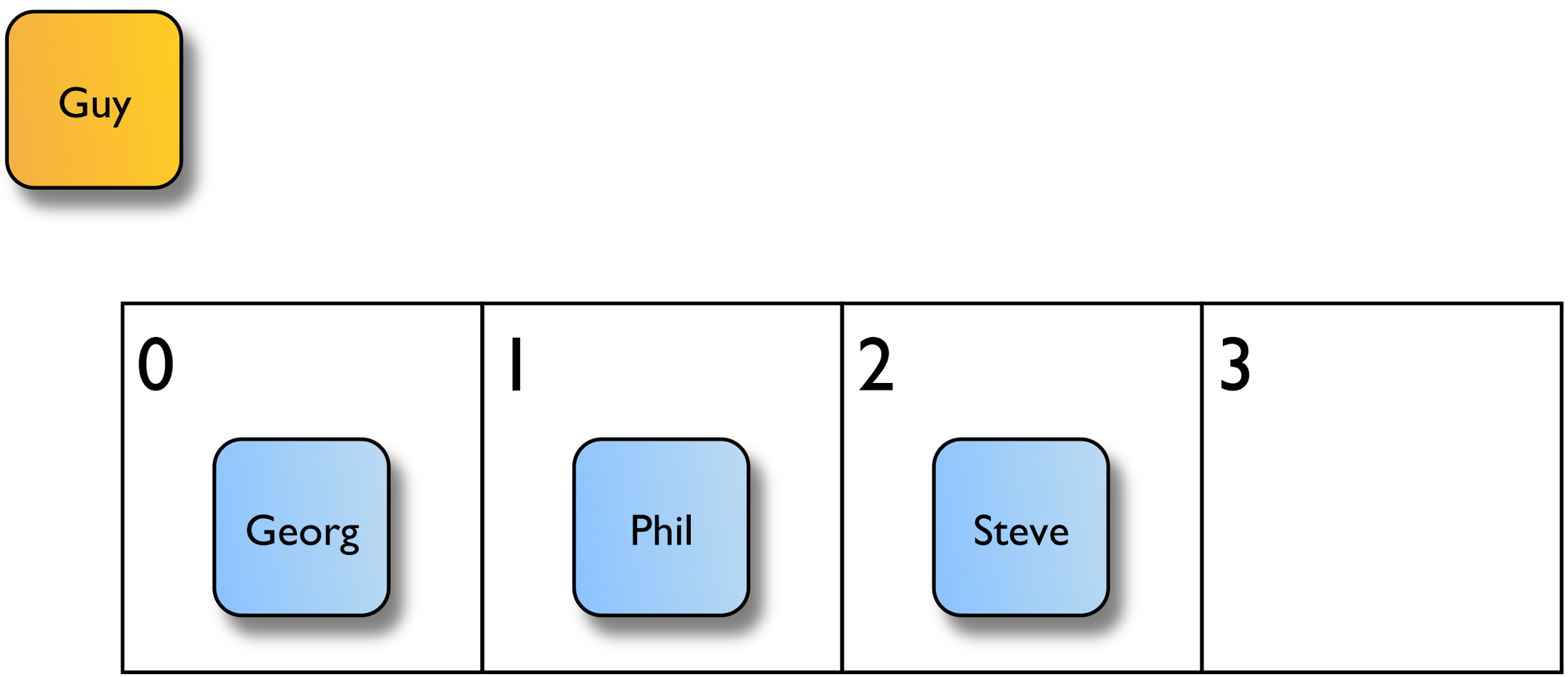
- If the internal array becomes too small, a bigger one is created, and all the objects copied across

ArrayList gives us

- Fast access to elements (as with arrays)
- Slow insertion into the middle of the list
- Slow deletion in middle of list

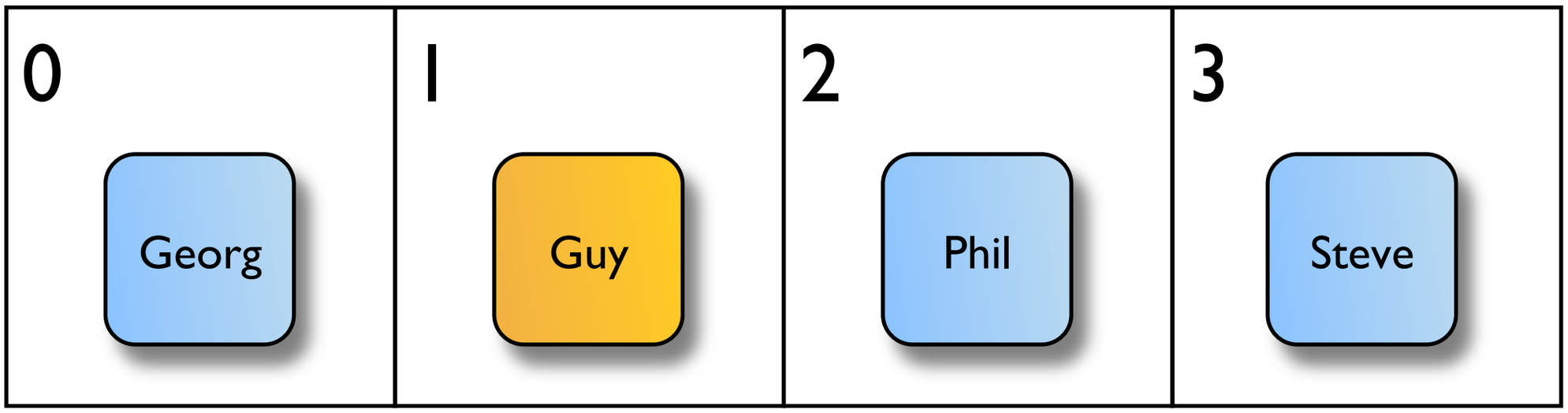
ArrayList - Slow Insertion

ArrayLists are implemented with arrays, so the shunting up of elements still has to occur, even though it happens "behind the scenes". This operation is slow when the list is big and the insertion is in the middle of the list.



ArrayList - Slow Deletion

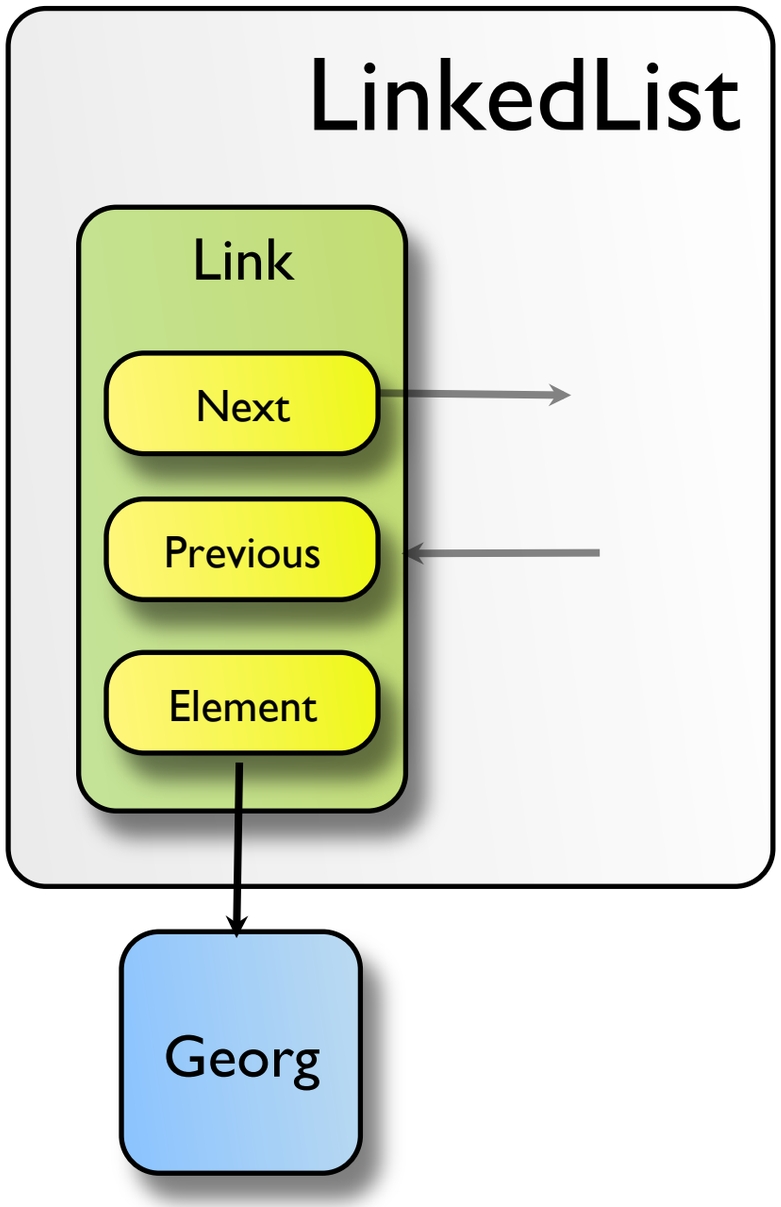
As with deletion, the shunting down of elements still has to occur with the internal array. This operation is slow when the list is big and the deletion is in the middle of the list.



LinkedList

LinkedList is an alternative implementation of a List.

Also implements List interface.



The list is made up of Link objects.

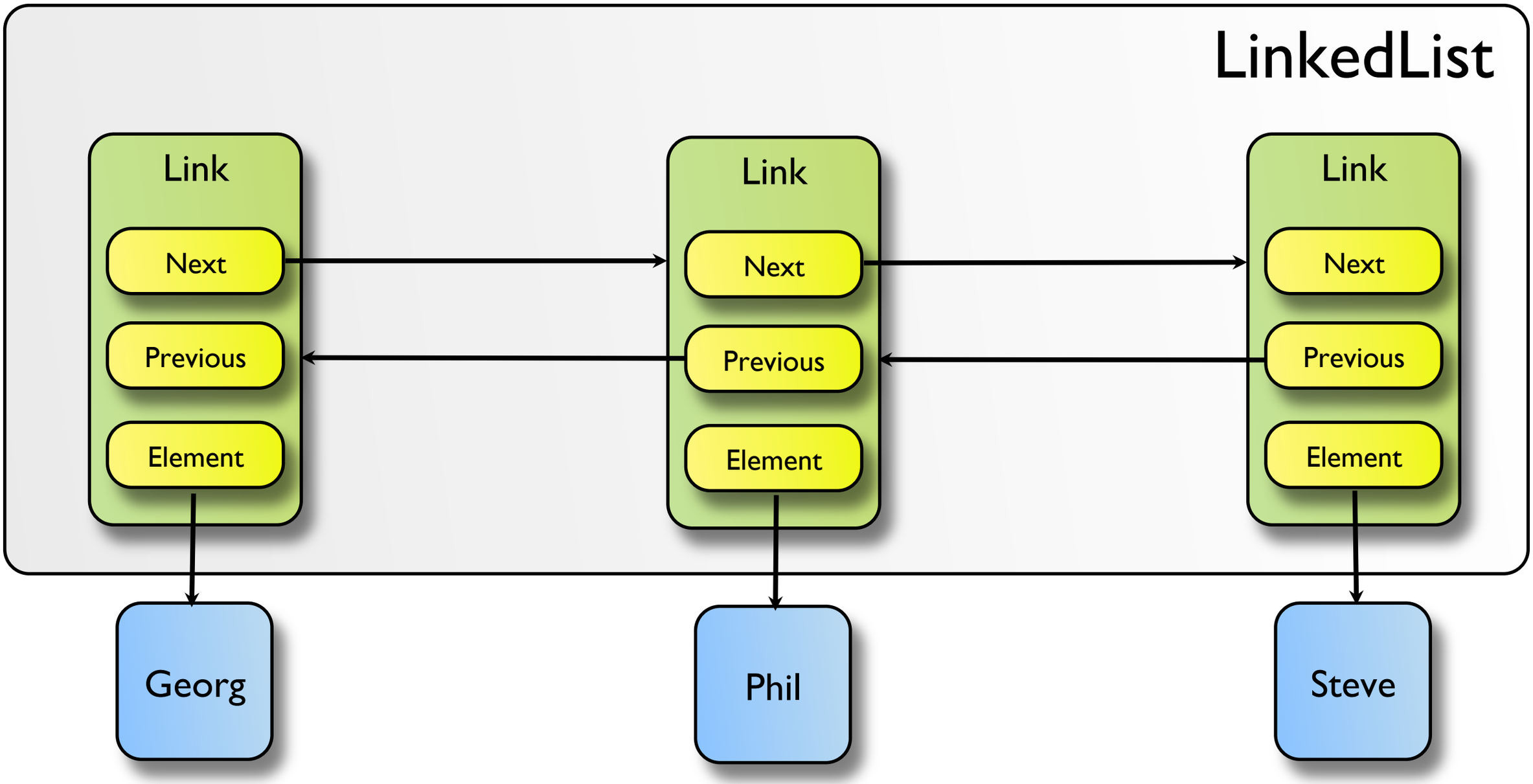
Each link object has a reference to an element in the list.

It also has a reference to the next link in the list (if there is another element in the list after this one)

... and the previous link in the list (if there is a previous element)

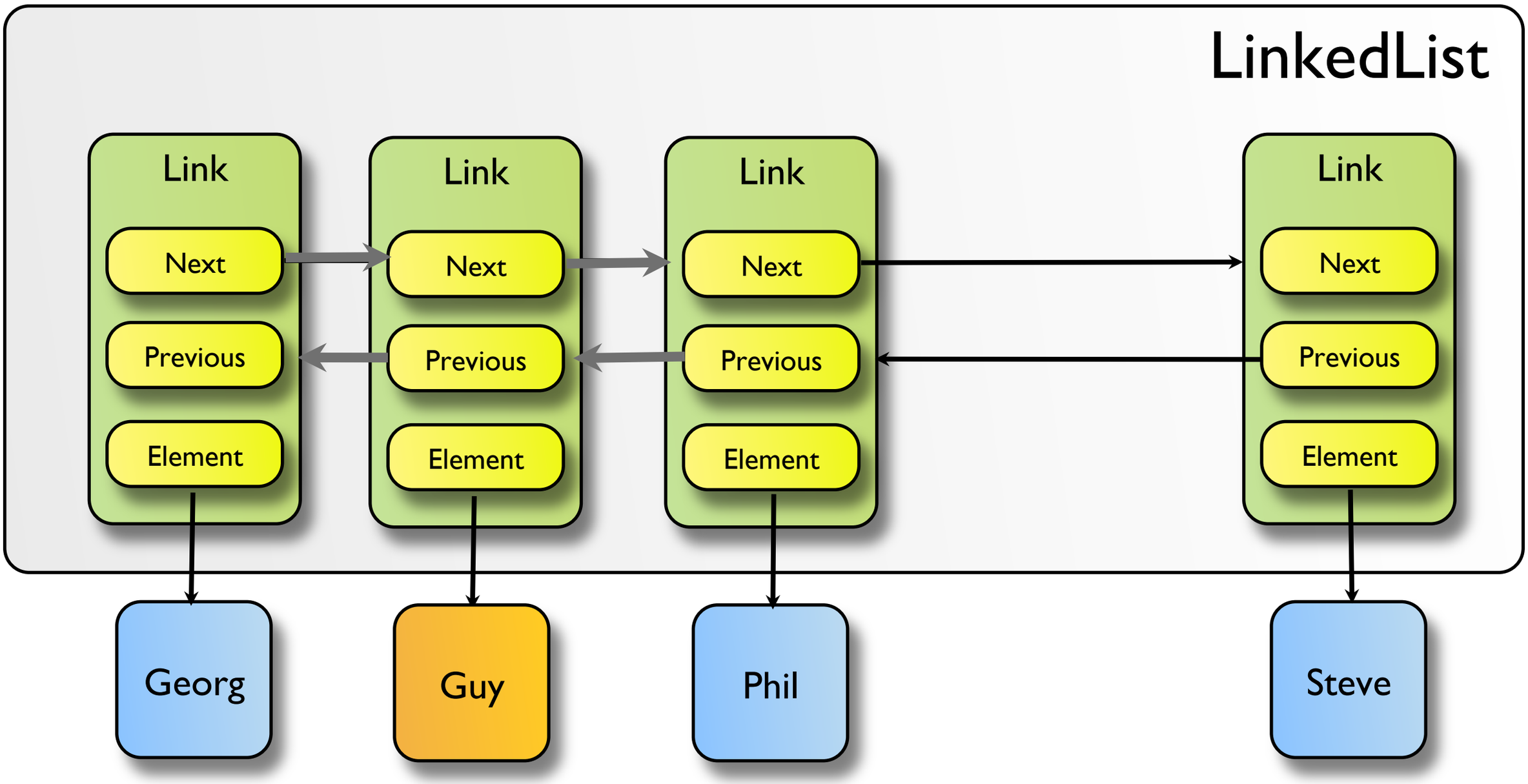
LinkedList

The links are chained together.



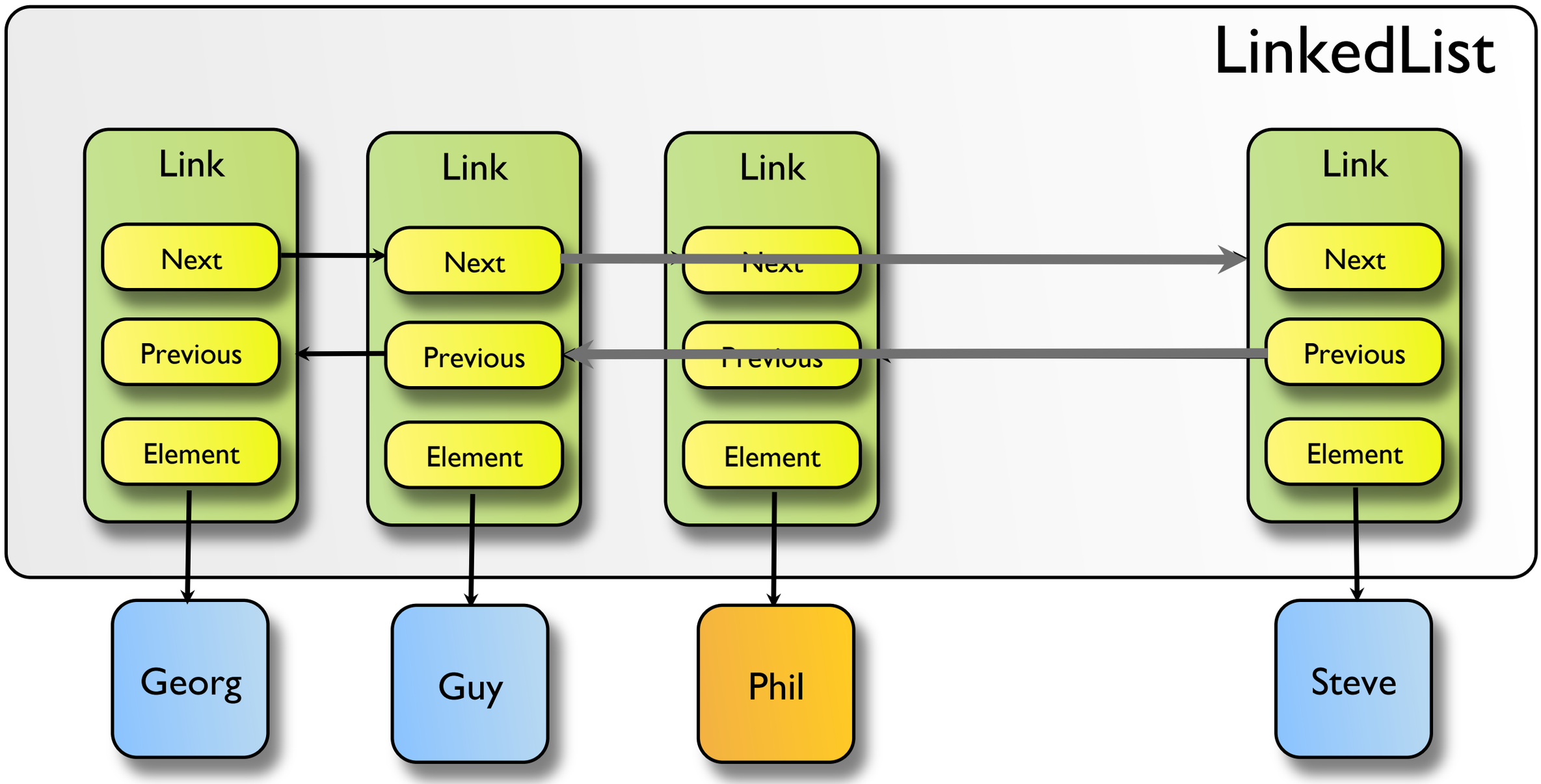
Linked List - Fast insertion

No need to move elements about, just create a new link and update the references in the links adjacent to it.



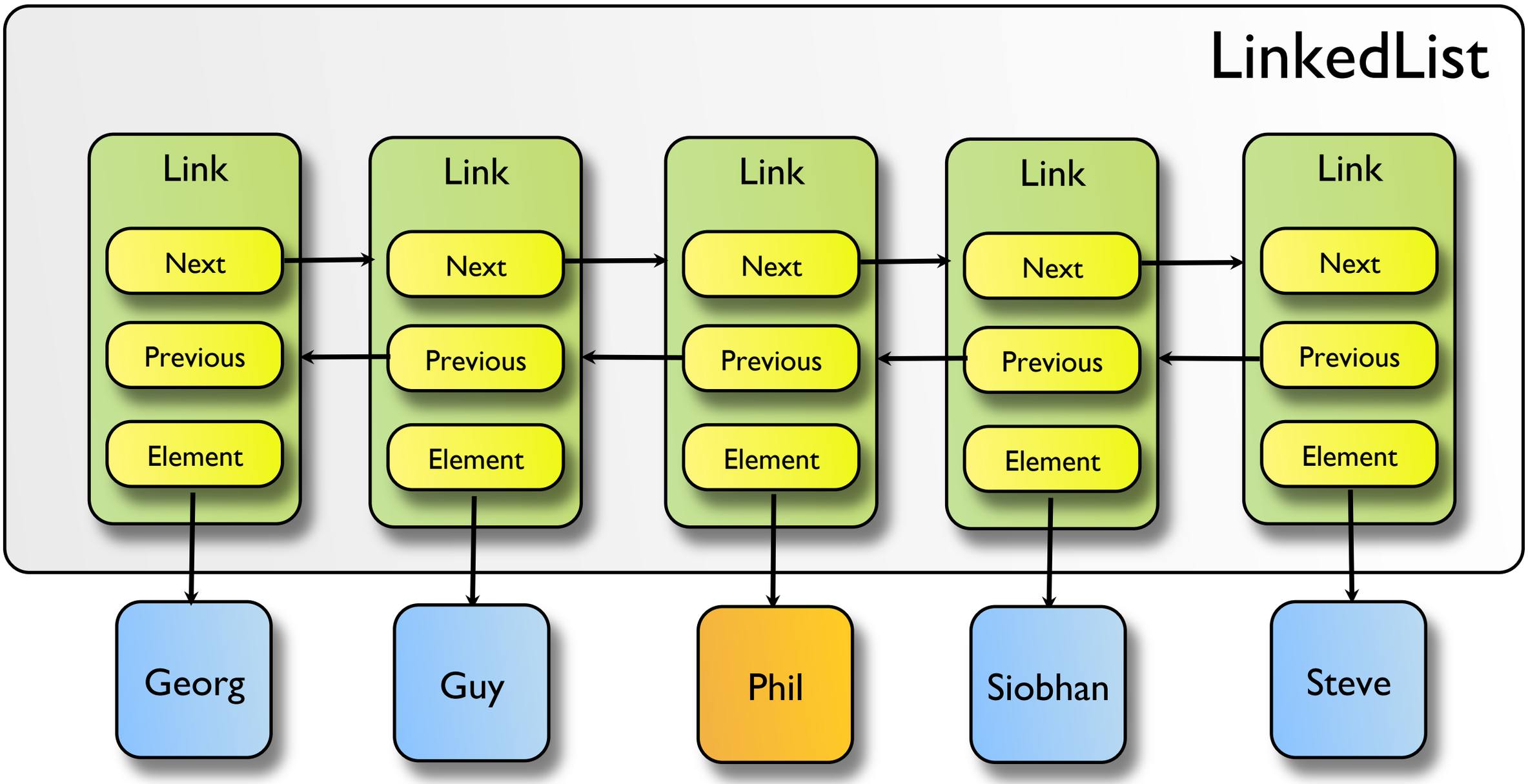
LinkedList - Fast deletion

No need to move elements about, just create a new link and update the references in the links adjacent to it.



Linked List - Slow Access

Conversely, access is slow, because we have to traverse the list from the beginning or end to get to the element we want.



ArrayList vs LinkedList

	ArrayList	LinkedList
Access	Fast (constant time)	Slow (scales with length of list)
Insertion	Slow (scales with length of list)	Fast (constant time*)
Deletion	Slow (scales with length of list)	Fast (constant time*)

* assuming you have already iterated to the place of insertion/deletion

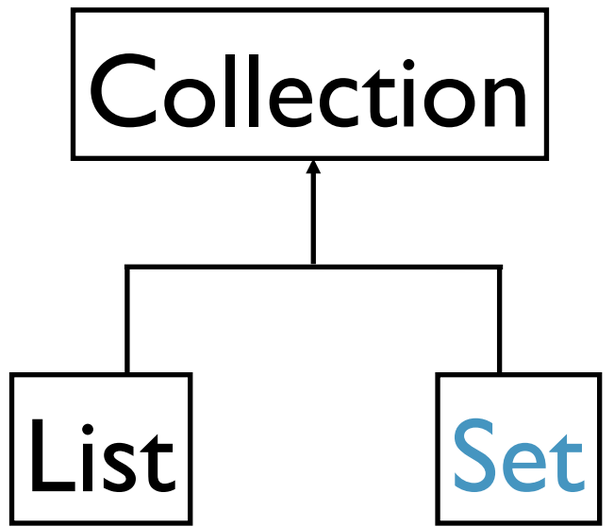
General Rule

ArrayList is faster than LinkedList,
except when you insert or remove an element
in the middle of the list.

Sets

Properties of a Set

- Its elements are unordered
- Its elements may not be duplicated



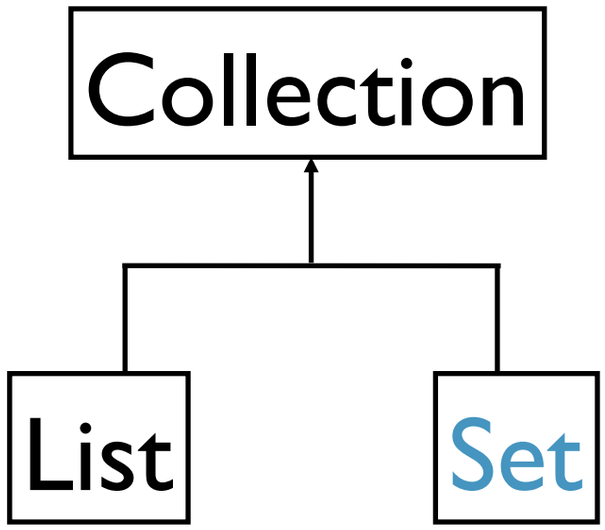
Set is implemented by

- TreeSet
- HashSet
- LinkedHashSet

All classes and interfaces live in java.util

The Set Interface

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

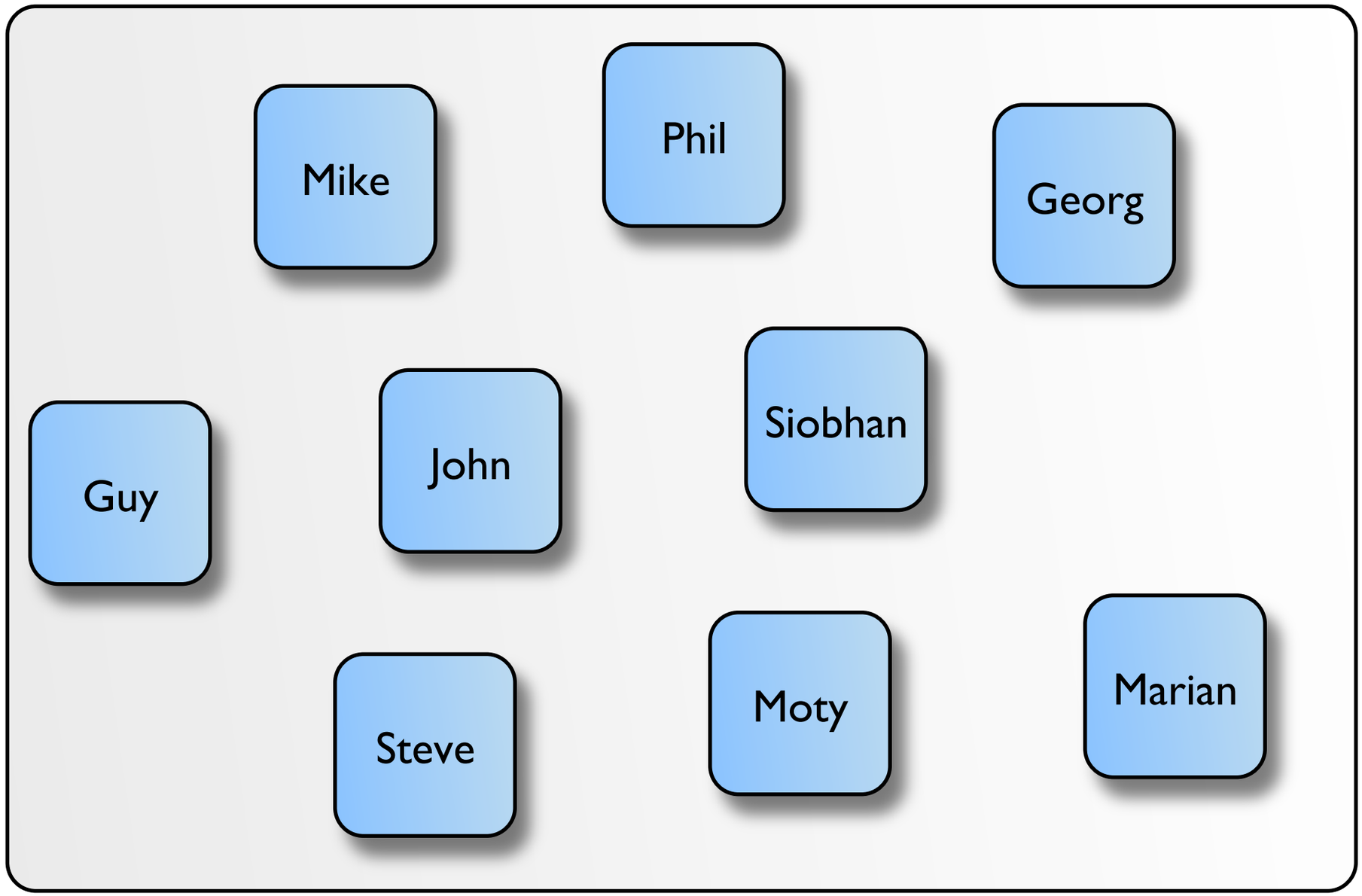


The Set interface is identical to that of Collection.

(The implementation has to differ from that for a List however, e.g. add must ensure the element is not already in the Set.)

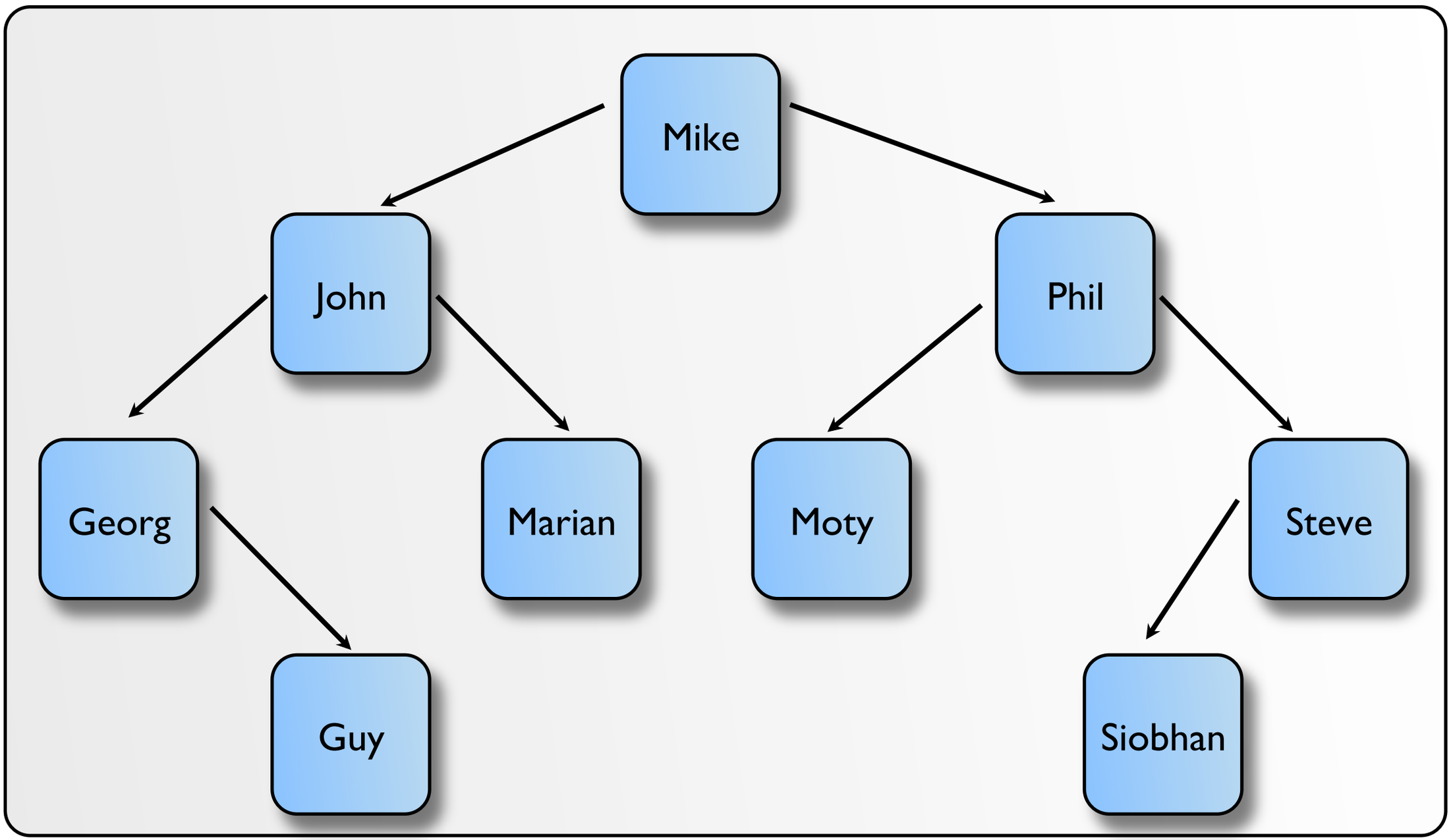
TreeSet

For sets, the inclusion test (whether an element is present in the set) is important, so it needs to implemented so that it is as fast as possible.



TreeSet - Implementation

TreeSet works to keep all elements in an ordered tree structure. The tree can be quickly searched to test for the presence/non-presence of an element.



Specifying Order with Comparators

Ordering is a common operation with collections.

The JCF requires us to implement either the Comparable or the Comparator interfaces to specify an ordering over elements.

Since TreeSet is dependent on ordering, the elements we put into it are reliant on one of these two interfaces.

Comparable specifies a single compareTo method, and Comparator specifies a more extensive set of methods. See the Java documentation for details.

Comparable

The compareTo method of Comparable compares this with some other object. It should return an int which is:

negative, if this < other

zero, if this = other

positive, if this > other

In the Lecturer class, we fall back on the compareTo method of String which itself implements compareTo with an alphabetical ordering:

```
public class Lecturer implements Comparable<Lecturer>{

    private String name;

    // ...

    public int compareTo(Lecturer other) {
        return this.getName().compareTo(other.getName());
    }
```

We have to specify the class name again in angle brackets when implementing Comparable

Here we just use the result of String's compareTo method on the names of the two respective objects.

The result of compareTo methods is often easily computable e.g. a - b for comparing two integers a and b

Comparator

The compare method of Comparator compares some object with some other object. It should return an int which is:

negative, if some < someOther

zero, if some = someOther

positive, if some > someOther

We have to specify the class name again in angle brackets when implementing Comparator

Here we fall back on String's compareTo method again, this time using surnames

```
public class SurnameComparator implements Comparator<Lecturer> {

    public int compare(Lecturer someLecturer, Lecturer someOtherLecturer) {
        // use string's comparator on the lecturer's surnames
        return someLecturer.getSurname().compareTo(someOtherLecturer.getSurname());
    }
}
```

Using TreeSet

Either we construct a TreeSet with a Comparator, or we leave it out and have the elements implement Comparable. If neither of these happen, the TreeSet will not be able to order the elements within it, and an exception will be thrown.

```
Set<Lecturer> lecturers = new TreeSet<Lecturer>(new SurnameComparator());

lecturers.add(new Lecturer("Georg Struth"));
lecturers.add(new Lecturer("Moty Katzman"));
lecturers.add(new Lecturer("Guy Brown"));
lecturers.add(new Lecturer("Joab Winkler"));
lecturers.add(new Lecturer("Phil McMinn"));

System.out.println(lecturers.contains(new Lecturer("Phil McMinn")));
System.out.println(lecturers.contains(new Lecturer("Mike Holcombe")));
```

HashSet

HashSets do not explicitly order elements, instead they are arranged for fast lookup using a data structure called a hash table.

A hash table contains a series of 'buckets'.

Each object needs to have a hashCode method, computed using its instance variables. The hashCode method returns a value which is used to decide which 'bucket' it should go in.

This results in fast access, so long as objects have unique hash codes and there are lots of buckets.

- Otherwise there will be a lot of 'collisions' - where several objects are placed in the same bucket.
- A collision means that each element in the bucket needs to be examined, one by one.

Computing Hash Codes

The hashCode method is defined in Object, so all objects have it by default. However, the default method does not give good hash codes.

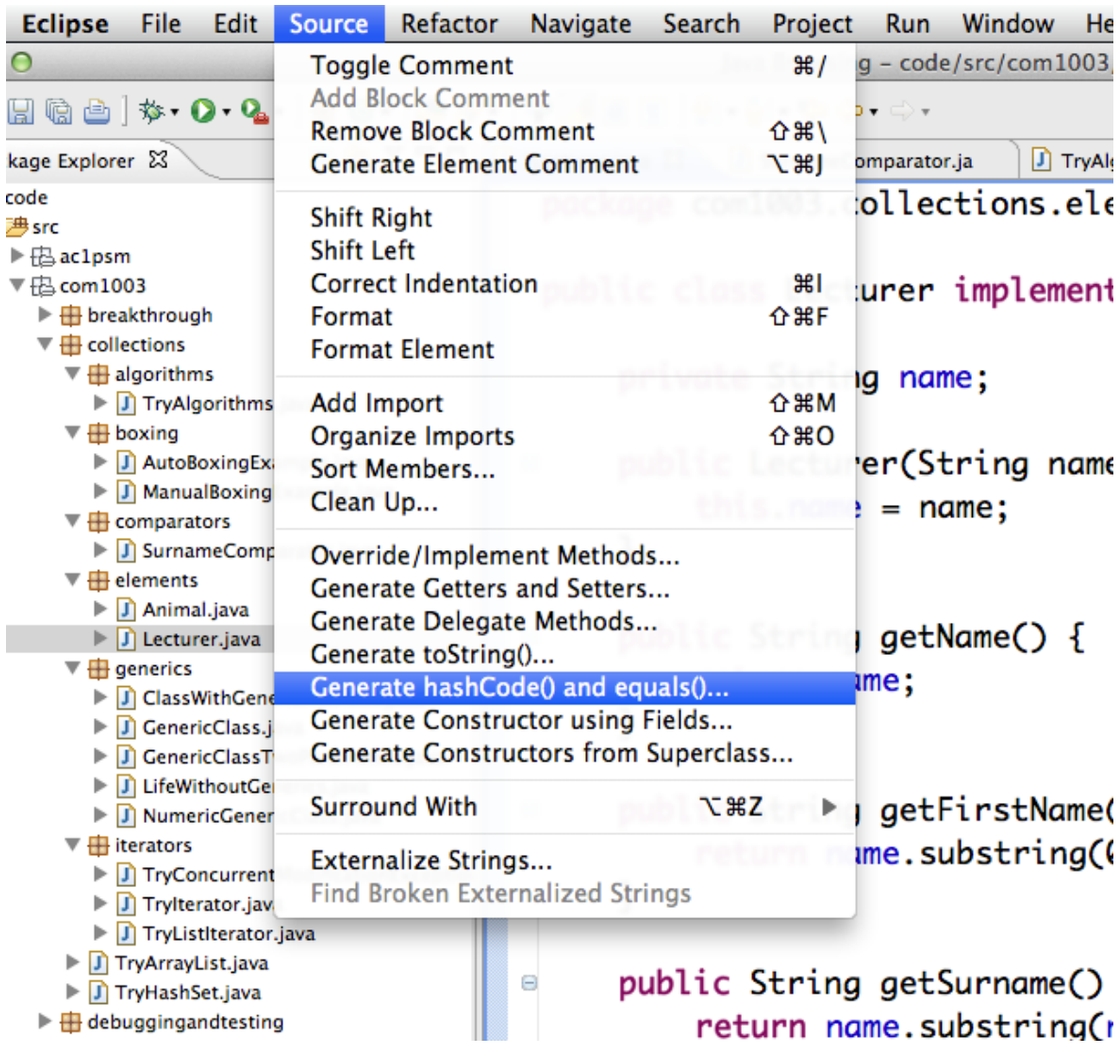
When defining your own classes, and you're using HashSet (or later - HashMap) override hashCode.

It needs to compute an integer that is more or less unique for the values of the object's instance variables.

The object's equals method also needs to be overridden so the hash table knows when an object (of the same class with identical state) has already been inserted into the table.

It follows that if x.equals(y) then x.hashCode() == y.hashCode()

Using Eclipse to override hashCode and equals



Open your class, go to the Source menu and click "Generate hashCode() and equals()".

Using Eclipse to override hashCode and equals

```java
public class Lecturer implements Comparable<Lecturer>{

    private String name;

    // ...

    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Lecturer other = (Lecturer) obj;
        if (name == null) {
            if (other.name != null) return false;
        } else if (!name.equals(other.name)) return false;
        return true;
    }
```

hashCode and equals generated for the Lecturer class.

Eclipse generates good hash codes based on prime numbers, which generate more distributed bucket numbers and thus reduce the number of collisions.

Hash tables and house keeping

When a HashSet is constructed we can specify the initial capacity (number of buckets) and the load factor.

The load factor determines when a re-hash should be performed.

- When the buckets start to fill up, to reduce the number of collisions a new hash table is created with a larger number of buckets, and each element is re-assigned a new bucket.

Good rules of thumb:

- The initial capacity should be 1.5 times the number of expected elements.
- The load factor is 0.75 (re-hash when 75% full)

If these aren't set, the default is 16 buckets and a load factor of 0.75.

Using HashSet

```
Set<Lecturer> lecturers = new HashSet<Lecturer>();

lecturers.add(new Lecturer("Georg Struth"));
lecturers.add(new Lecturer("Moty Katzman"));
lecturers.add(new Lecturer("Guy Brown"));
lecturers.add(new Lecturer("Joab Winkler"));
lecturers.add(new Lecturer("Phil McMinn"));

for (Lecturer l : lecturers) {
    System.out.println(l.getName() +
                       " is in the set and his hash code is "+
                       l.hashCode());
}
```

TreeSet or HashSet ?

Lookup times for TreeSet grow as the number of elements in it increase.

Lookup times for HashSet remain more or less constant (assuming good hash codes and sufficient buckets to reduce collisions).

Therefore, HashSet is generally a better choice than TreeSet.

But of course, remember:

- HashSet requires elements have a suitable hashCode and equals method
- TreeSet requires elements implement Comparable or has a Comparator (i.e. a natural order exists amongst elements)

Maps

Accessing elements using “keys”

A set is a collection that lets you find an existing element quickly.

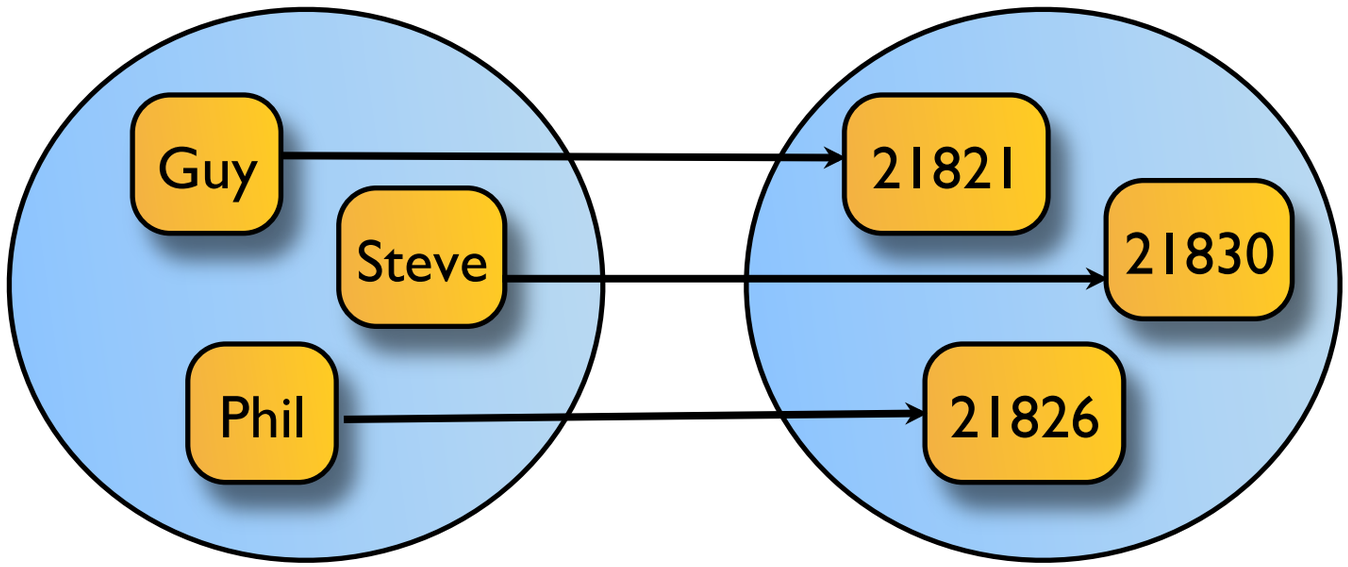
But to look up an element:

- You need a copy of it (i.e. you have all the information already), or
- You need to iterate through the entire set (slow)

It is much more common to retrieve a value using a key, e.g.

- A student record using a UCard number.
- Someone’s phone number, given their name.
- The number of times a word appears in a document, given a word.

Maps



Lecturers

University Telephone extension numbers

Keys

Values

Properties of a Map

Properties of a Map:

- Each key maps to exactly one value
- The map may contain duplicate values, but it cannot contain duplicate keys.

The Map interface in Java is implemented by

- TreeMap
- HashMap

All classes and interfaces live in java.util

The Map interface

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
}
```

Two class placeholders.

K represents the class of the keys,

V the class of the values

Note that Map, unlike List and Set, *does not* implement Collection.

Implementations of Map

The set of a keys is implemented with a Set.

The two main implementations of Set are TreeSet and HashSet. Thus:

- TreeMap uses a TreeSet for managing keys, while
- HashMap uses a HashMap for managing keys.

As for sets:

HashMap is generally more efficient than TreeMap.

and the same pre-requisites apply for both:

TreeSet requires elements implement Comparable or it is constructed with a Comparator.

HashMap requires hashCode and equals to be overridden by elements.

Using HashMap

```
Map<Lecturer, String> phoneBook = new HashMap<Lecturer, String>();

phoneBook.put(new Lecturer("Phil"), "21826");
phoneBook.put(new Lecturer("Guy"),  "21821");
phoneBook.put(new Lecturer("Fax"),  "21810");

// get Guy's phone number
System.out.println(phoneBook.get(new Lecturer("Guy")));

// remove Fax
System.out.println(phoneBook.remove(new Lecturer("Fax")));

// update Phil's number
phoneBook.put(new Lecturer("Phil"), "21827");
```

Summary

Java Collections:

Lists

ArrayList, LinkedList

Sets

TreeSet, HashSet

Maps

TreeMap, HashMap

ConcurrentModificationException

Beware: Iterators are jealous creatures.

Java gets upset if you mess with the contents of a collection when its iterating over it - unless you're using that iterator to actually do the adding or removing as shown before.

```
Iterator<Lecturer> iterator = lecturers.listIterator();

while (iterator.hasNext()) {
    Lecturer lecturer = iterator.next();

    if (lecturer.getName().equals("Phil McMinn")) {
        // Add an element to the end of the list.
        // We're not doing this using iterator, but by
        // using the list directly.  This will trigger
        // a ConcurrentModificationException and crash
        // the program.
        lecturers.add(new Lecturer("Mike Holcombe"));
    }

    System.out.println(lecturer.getN
}
```

```
java.util.ConcurrentModificationException
        at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372)
        at java.util.AbstractList$Itr.next(AbstractList.java:343)
        at com1003.collections.iterators.TryConcurrentModificationException.main(TryConcurrentModificationException.
```

Iterating through the elements of a Map

Maps do not implement Collection and do not have the iterator method.

```
Map<Lecturer, String> phoneBook = new HashMap<Lecturer, String>();

phoneBook.put(new Lecturer("Phil"), "21826");
phoneBook.put(new Lecturer("Guy"),  "21821");
phoneBook.put(new Lecturer("Fax"),  "21810");

// ...
```

However, we can get the Map's key set and iterate through that instead to enumerate the elements, if we wish.

```
// iterate using the map's key set iterator
Iterator<Lecturer> iterator = phoneBook.keySet().iterator();
while (iterator.hasNext()) {
    Lecturer l = iterator.next();
    System.out.println(l.getName()+"'s phone extension is "+phoneBook.get(l));
}

// iterate "for-each" style:
for (Lecturer l : phoneBook.keySet()) {
    System.out.println(l.getName()+"'s phone extension is "+phoneBook.get(l));
}
```

Algorithms

Algorithms in the JCF

- Data manipulation
- Sorting
- Searching
- Shuffling
- Finding extreme values

Algorithms are implemented in the
java.util.Collections class

Data Manipulation in the Collections class

With List:

- reverse method reverses list order
- fill overwrites very element with a specified object
- copy overwrites elements of a destination list with a source list
- swap swaps two elements at specified positions

With Collection:

- addAll adds elements from another collection

Sorting

The sort method sorts a list.

(The algorithm used is called “merge sort”)

Default: use compareTo in the Lecturer class:
(sort on whole name)

```
Collections.sort(lecturers);
```

```
0) Georg Struth
1) Guy Brown
2) Joab Winkler
3) Marian Gheorghe
4) Mike Stannett
5) Moty Katzman
6) Phil McMinn
7) Siobhan North
8) Steve Maddock
```

Using a supplied comparator:
(sort on surname)

```
Collections.sort(lecturers, new SurnameComparator());
```

```
0) Guy Brown
1) Marian Gheorghe
2) Moty Katzman
3) Steve Maddock
4) Phil McMinn
5) Siobhan North
6) Mike Stannett
7) Georg Struth
8) Joab Winkler
```

Searching

The binarySearch method returns the index of an element in a sorted list.

```
// list must be in sorted order
Collections.sort(lecturers);

int index = 0;
for (Lecturer l : lecturers) {
    System.out.println(index + ") "+ l.getName());
    index ++;
}

String name = "Phil McMinn";
int nameIndex = Collections.binarySearch(lecturers, new Lecturer(name));

System.out.println(name+" is at index "+nameIndex);
```

```
0) Georg Struth
1) Guy Brown
2) Joab Winkler
3) Marian Gheorghe
4) Mike Stannett
5) Moty Katzman
6) Phil McMinn
7) Siobhan North
8) Steve Maddock
Phil McMinn is at index 6
```

If a negative value is returned, the element is not present.

Requires efficient access to work well, i.e. an ArrayList.

Shuffling

```
Sorted list:
0) Georg Struth
1) Guy Brown
2) Joab Winkler
3) Marian Gheorghe
4) Mike Stannett
5) Moty Katzman
6) Phil McMinn
7) Siobhan North
8) Steve Maddock
```

The shuffle method rearranges a list into random order.

```
Collections.shuffle(lecturers);
```

```
Shuffled list:
0) Joab Winkler
1) Georg Struth
2) Siobhan North
3) Marian Gheorghe
4) Mike Stannett
5) Guy Brown
6) Moty Katzman
7) Phil McMinn
8) Steve Maddock
```

Finding Extremal Values

```
// list does not need to be sorted
Collections.shuffle(lecturers);

Lecturer first = Collections.min(lecturers);
System.out.println(first.getName());
```

Find the minimum in the list according to the comparator (none is supplied, so is using compareTo in Lecturer - which compares the whole name).
This will print the lecture that comes first by name in alphabetical order.

```
// list does not need to be sorted
Collections.shuffle(lecturers);

Lecturer last = Collections.max(lecturers, new SurnameComparator());
System.out.println(last.getName());
```

Find the maximum in the list according to the comparator (SurnameComparator is supplied, which compares by surname whole name).
This will print the lecture that comes last by surname in alphabetical order.

Generics

Recall the angled brackets...

```java
import java.util.ArrayList;

// ...

ArrayList<String> lecturers = new ArrayList<String>();

lecturers.add("Georg");
lecturers.add("Phil");
lecturers.add("Steve");

// add Guy at position 1
lecturers.add(1, "Guy");

// remove Guy at position 1
lecturers.remove(1);
```

```java
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
```

Life without Generics

```
ArrayList list = new ArrayList();

list.add(new Lecturer("Phil McMinn"));

Lecturer lecturer = (Lecturer) list.get(0);
System.out.println(lecturer.getName());

Animal animal = (Animal) list.get(0);
System.out.println(animal.getName());
```

If we don't supply the angled brackets when instantiating a collection, Java will assume the class is Object

This is how Java was before version 5.0. However, it involved a lot of tiresome casting to get objects back to the intended type.

... and, as we know - casting can be unsafe. This code will compile, but will crash at runtime

```
Exception in thread "main" java.lang.ClassCastException: com1003.collections.elements.Lecturer cannot be cast to com1003.collections.elements.Animal
        at com1003.collections.generics.LifeWithoutGenerics.main(LifeWithoutGenerics.java:20)
```

So what are Generics?

Generics allow classes to be "parameterised".

They enable the creation of classes, interfaces and methods in which certain classes are instead specified by a placeholder called a class parameter.

Equivalent code can be written without Generics, but using Generics:

- Simplifies code - no need for tiresome casts.
- Adds type safety to our code.

A Simple Example

Class Parameter used through the class definition

(must appear here after the class name)

```
public class GenericClass<T> {

    T obj; // declare an instance variable of class T

    public GenericClass(T obj) {
        this.obj = obj;
    }

    T getObj() {
        return obj;
    }

    void printClass() {
        System.out.println("Class parameter is "+obj.getClass().getName());
    }
}
```

A Simple Example

```java
public class GenericClass<T> {

    T obj; // declare an instance variable of class T

    public GenericClass(T obj) {
        this.obj = obj;
    }

    T getObj() {
        return obj;
    }

    void printClass() {
        System.out.println("Class parameter is "+obj.getClass().getName());
    }
}
```

```java
GenericClass<Lecturer> l =
        new GenericClass<Lecturer>(new Lecturer("Phil McMinn"));
l.printClass();

GenericClass<Animal> a =
        new GenericClass<Animal>(new Animal("Dog"));
a.printClass();

GenericClass<Double> d =
    new GenericClass<Double>(0.42);
d.printClass();
```

prints

```
Class parameter is com1003.collections.elements.Lecturer
Class parameter is com1003.collections.elements.Animal
Class parameter is java.lang.Double
```

Two Parameters

```
public class GenericClassTwoParameters<T, V> {

    T obj1; // declare an instance variable of class T
    V obj2; // declare an instance variable of class V

    public GenericClassTwoParameters(T obj1, V obj2) {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    T getObj1() {
        return obj1;
    }

    V getObj2() {
        return obj2;
    }
}
```

Generic Methods

Methods may also involve class parameters that are placeholders for the classes of the method's parameters, e.g.

```
public <N> void print(N obj) {
    // print something to do with N
}
```

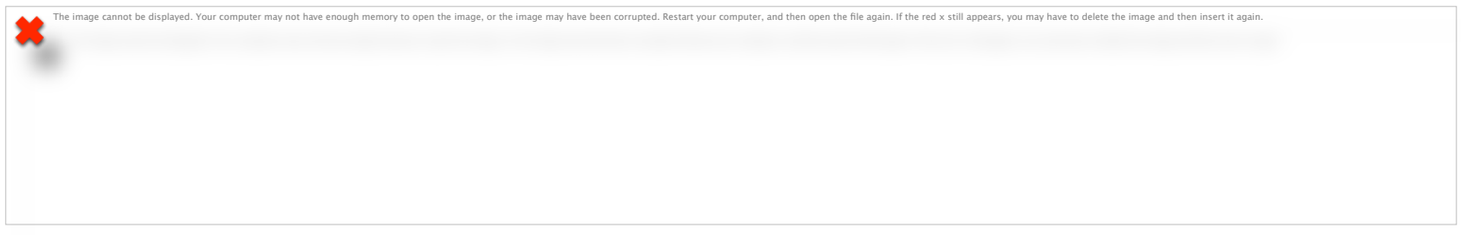
The class parameter must be declared in angle brackets just before the return type of the method.

Bounded Parameters

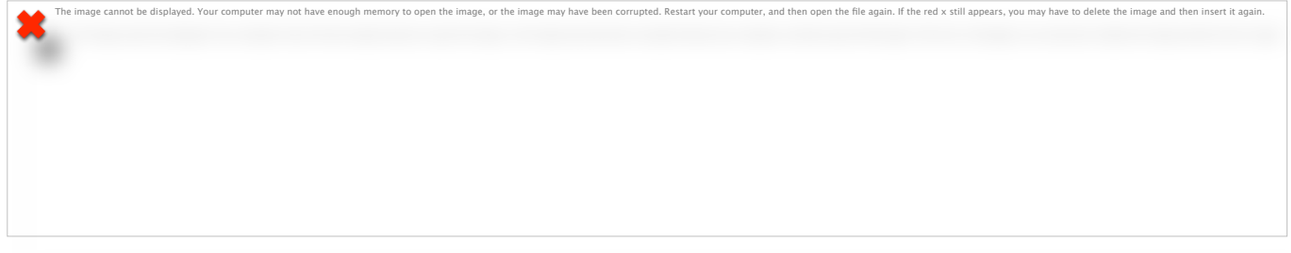
We can limit T to being a subclass of X:

<T extends X>

For example, to ensure the class is of a numeric type:
(Number is the superclass of Integer and Double)



Class Level



Method Level

Wildcards

If the generic parameter itself should specify a generic class, we can use wild cards to specify the parameters of that generic class.

(Method-level only)

```
public <N> void print(GenericClass<?> obj) {
    // print something
}

public <N> void print2(GenericClass<? extends Number> obj) {
    // print something
}
```

Boxing


BEDROOM
BOOK
16¾X12⅝X12⅞
1.5 CUBE
BEDROOM
MEDIUM
18 ½ X 16 X 16
3.0 CUBE
BEDROOM
EXTRA LARGE 6.0 CU. FT.

Classes not primitives

When the generic class is instantiated, the class parameter must be a class, not a primitive type. So:

```
ArrayList<int> l = new ArrayList<int>()
```

won't work, since int is a primitive type, not a class.

If we want it to work with integers we need the Integer class:

```
ArrayList<Integer> l = new ArrayList<Integer>()
```

Boxing and unboxing

Of course, that potentially leads to all sorts of messing about converting from primitive types to their associate classes and back.

```
ArrayList<Integer> l = new ArrayList<Integer>();
l.add(new Integer(5));

Integer boxedValue = l.get(0);
int unboxedValue = boxedValue.intValue();
```

The process of converting a primitive type to an object of its associated class is calling boxing.

The opposite process (converting an object to its associated a primitive type) is called unboxing.

Autoboxing

Boxing and unboxing is a pain.

Java will do it for you (without making a fuss).

```
ArrayList<Integer> l = new ArrayList<Integer>();
l.add(5);

int unboxedValue = l.get(0);
```

compare with
manual version

```
ArrayList<Integer> l = new ArrayList<Integer>();
l.add(new Integer(5));

Integer boxedValue = l.get(0);
int unboxedValue = boxedValue.intValue();
```

Autoboxing “gotchas”

Java’s autoboxing magic means you can largely ignore the distinction between primitive types and their class equivalents, except if you try to unbox a null element:

- References can be null (i.e. no object)
- But there is no null value for primitives.
- An attempt to unbox a null element will crash your program.

Java 7 Diamond operator

- The "diamond" operator (from Java 7 onwards) means we don't have to specify the class parameter during construction
- Instead of:

```
ArrayList<String> lecturers = new ArrayList<String>();
```

We can simply write:

Java 7 infers "String" from the left hand side.

```
ArrayList<String> lecturers = new ArrayList<>();
```

This applies to all classes using generics - see UsingDiamondOperator.java in the uk.ac.sheffield.com1003.collections package.