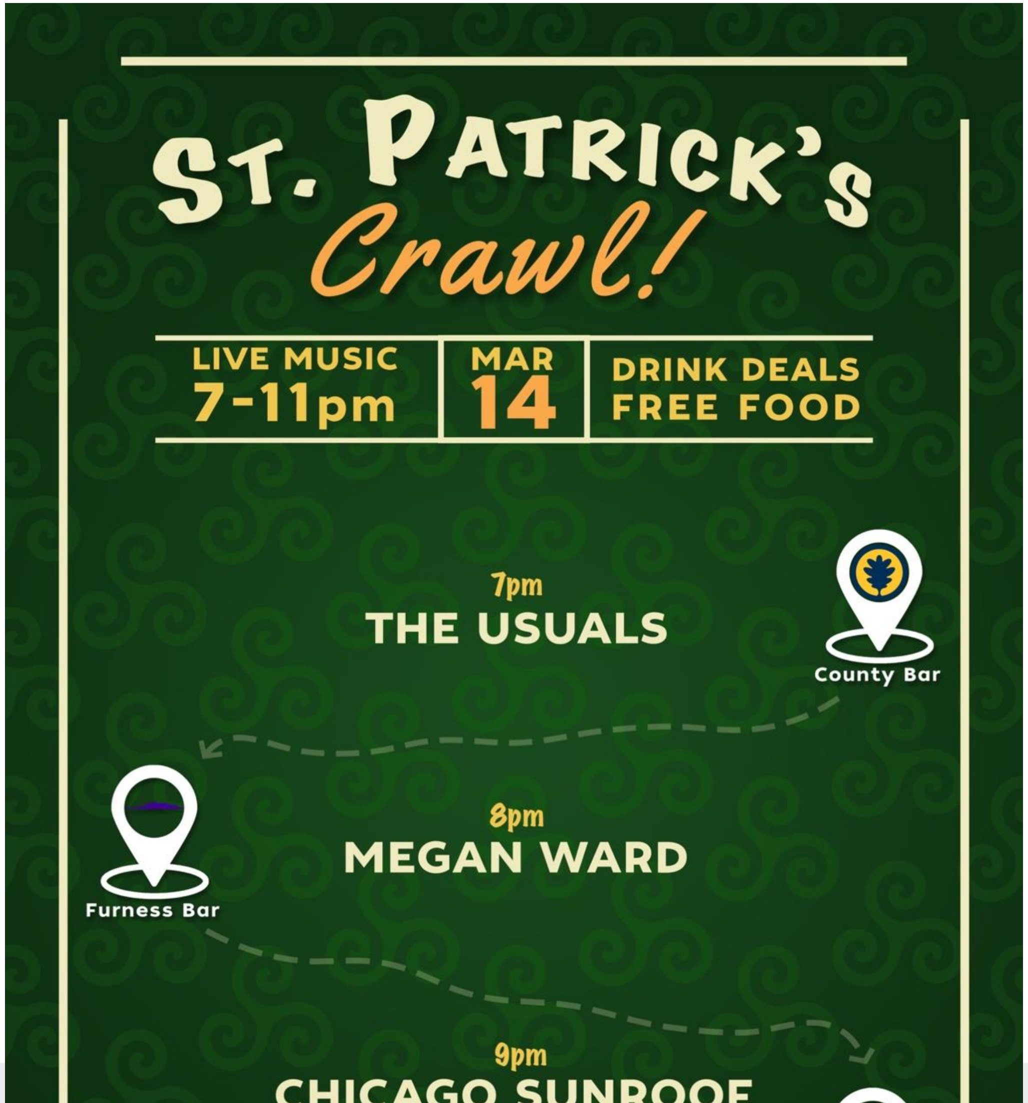
School of Computing & Communications | Lancaster University

SCC.111 Software Development – Lecture 38: The Collections API and Generics

Adrian Friday, Hansi Hettiarachchi and Nigel Davies

The Usuals

Lancaster University



St. Patrick's

Crawl!

LIVE MUSIC 7-11pm	MAR 14	DRINK DEALS FREE FOOD

7pm

THE USUALS

County Bar

8pm

MEGAN WARD

Furness Bar

9pm

CHICAGO SUNROOF

GameArena Showcase!

- We've started to see some great games being submitted to the **GameArenaSubmissions** repository!

- These are fantastic - please *keep them coming* and share your work.
- To submit, please just raise a **merge request** against the repo, with your work with all the code needed to run you program in a folder matching your project name.
- https://scc-source.lancs.ac.uk/scc.Y1/scc.111-workarea/gamearenasubmissions
- We'll review all those submitted by Wednesday Week 20.
- **We'll have some live demos, awards and prizes in the Friday Week 20 lecture!**

Introduction

- In the last lectures, we:
  - Revised the core concepts of Object-Oriented programming
  - Practiced these concepts with a case study on the GameArena API

- Today we will
  - Introduce the Collections API
  - Introduce something called Generics

Collections

Collections are just data structures

- They provide general, reusable implementations of common data structures.
- They are not inherently part of the language per se…

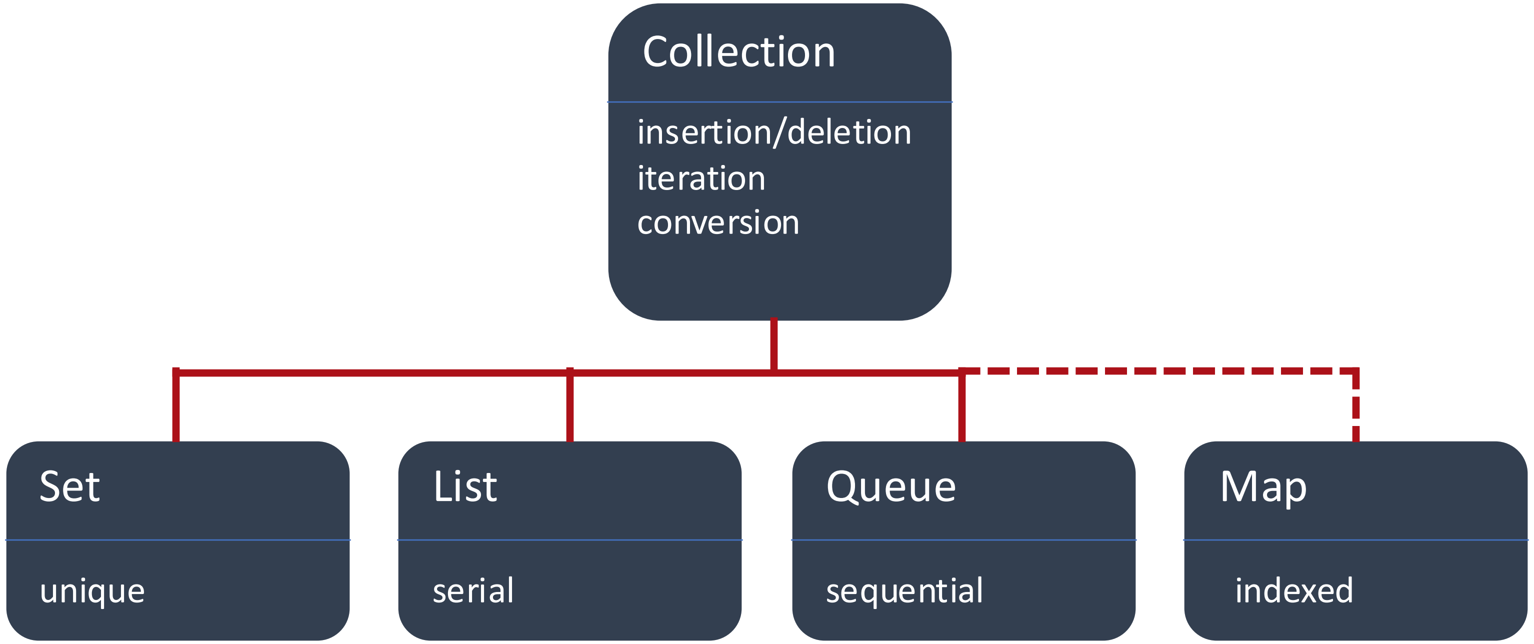
- They are simply implemented as classes
- Most popular data structures are distributed in the standard class library
- But others can be written if needed

- **Let's see what they look like…**

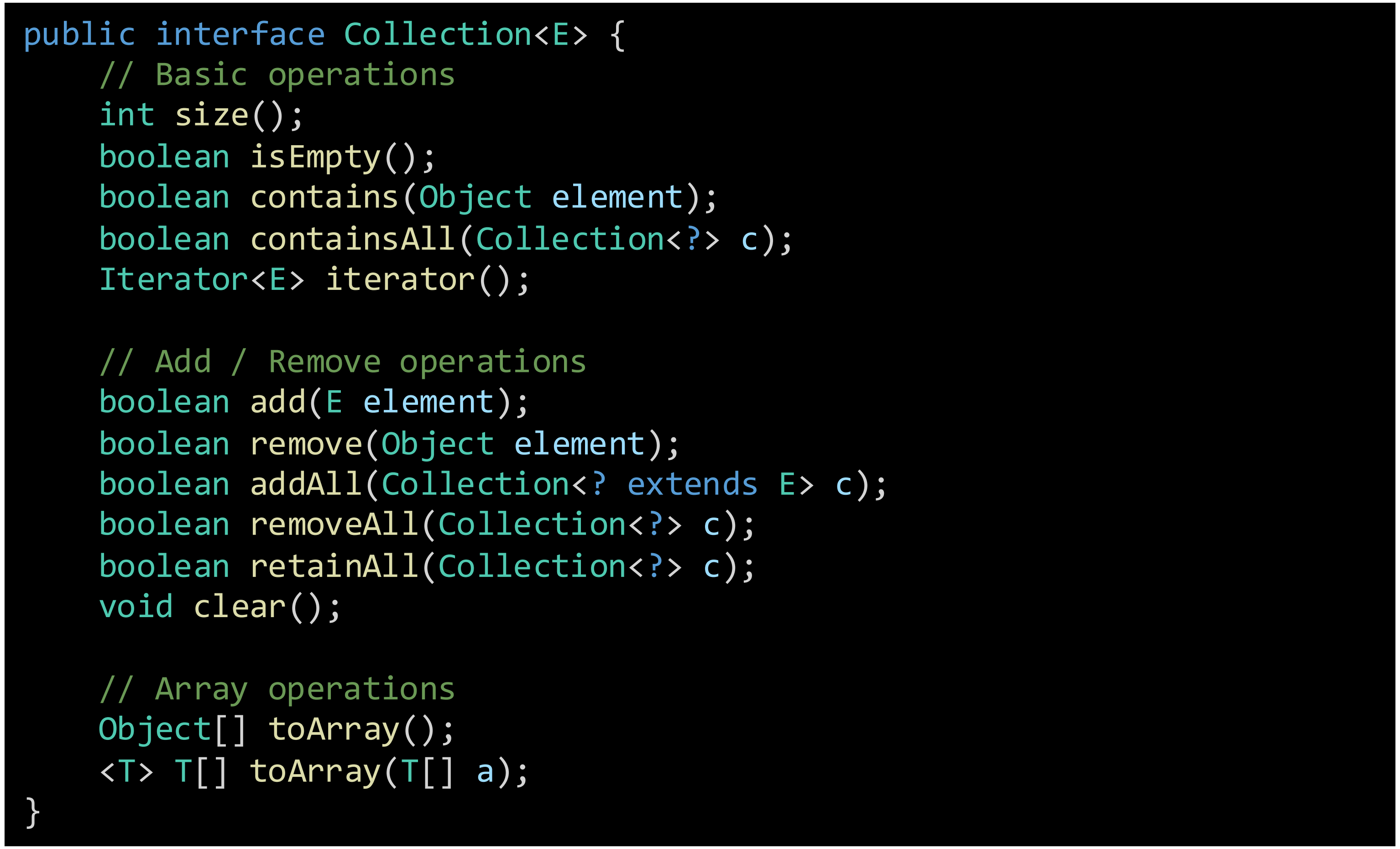
Collections Enable Abstract Data Types

Java Collections are implemented through an interface hierarchy

- Therefore, all data structures implementation have the same API
- Classes such as LinkedList implement these interfaces

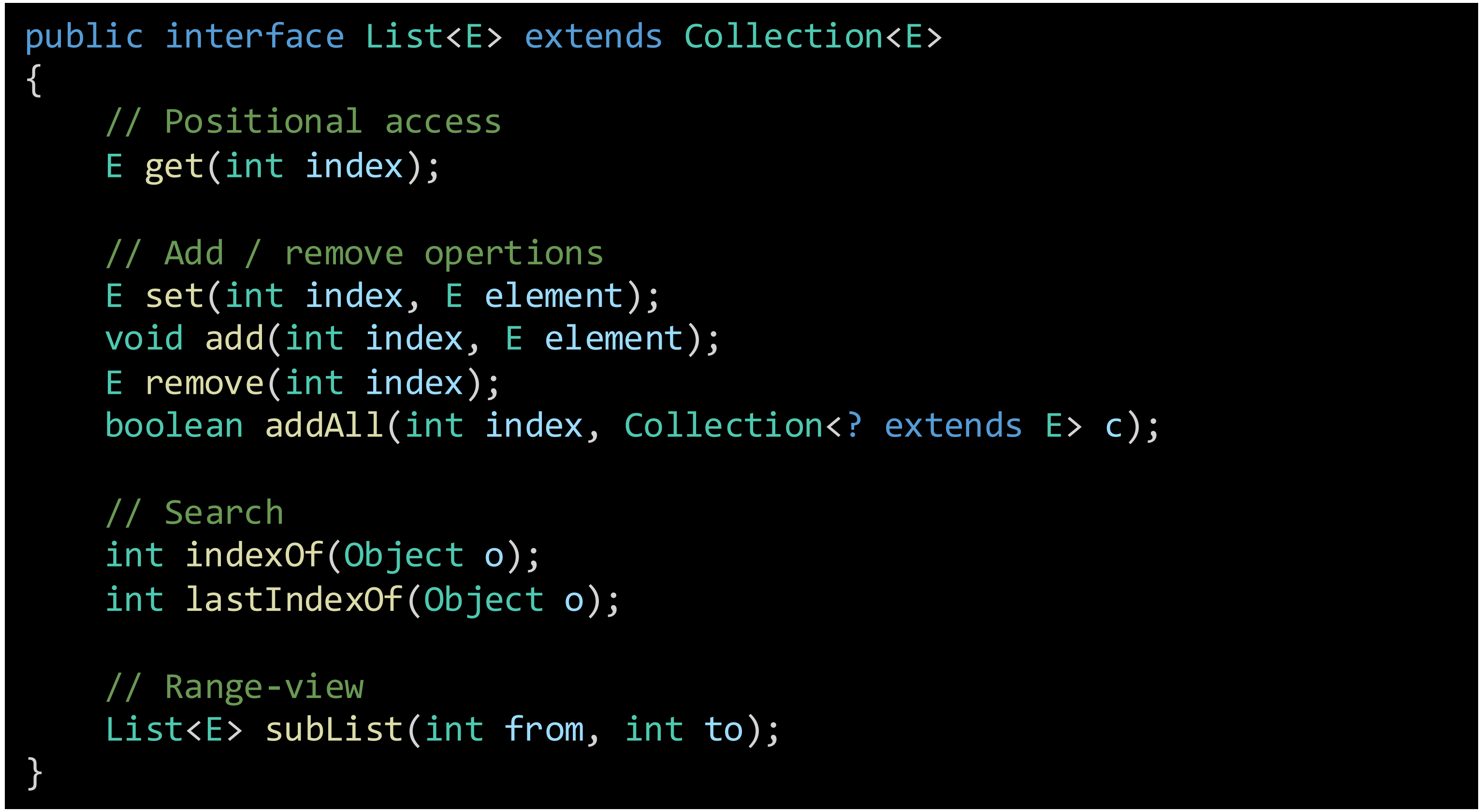


Collection Interface



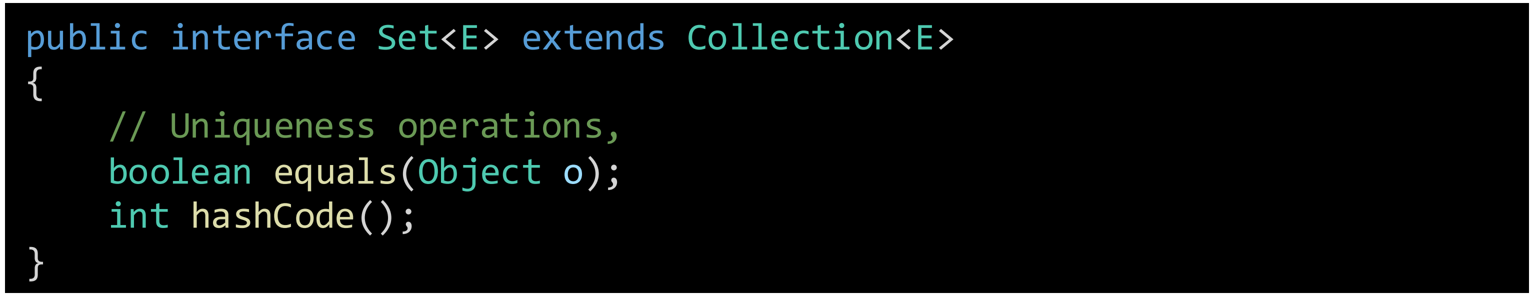
```
public interface Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean containsAll(Collection<?> c);
    Iterator<E> iterator();

    // Add / Remove operations
    boolean add(E element);
    boolean remove(Object element);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

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

List Interface

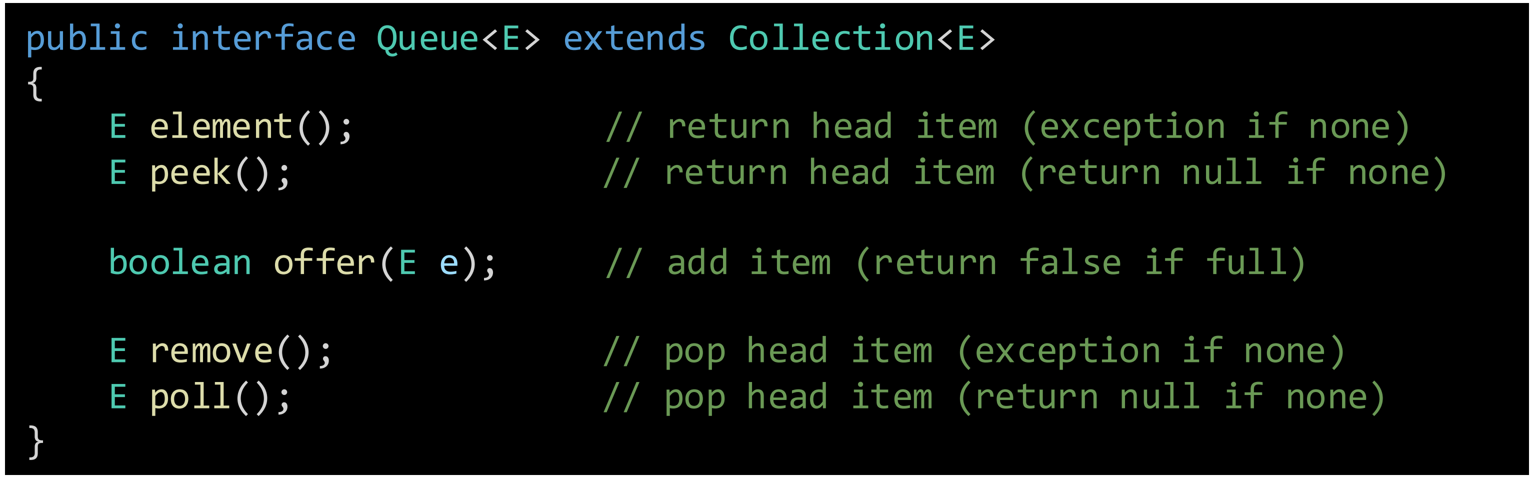


```
public interface List<E> extends Collection<E>
{
    // Positional access
    E get(int index);

    // Add / remove opertions
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    boolean addAll(int index, Collection<? extends E> c);

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

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

Set and Queue Interfaces



```
public interface Set<E> extends Collection<E>
{
    // Uniqueness operations,
    boolean equals(Object o);
    int hashCode();
}
```



```
public interface Queue<E> extends Collection<E>
{
    E element();            // return head item (exception if none)
    E peek();               // return head item (return null if none)

    boolean offer(E e);     // add item (return false if full)

    E remove();             // pop head item (exception if none)
    E poll();               // pop head item (return null if none)
}
```

Common Implementations

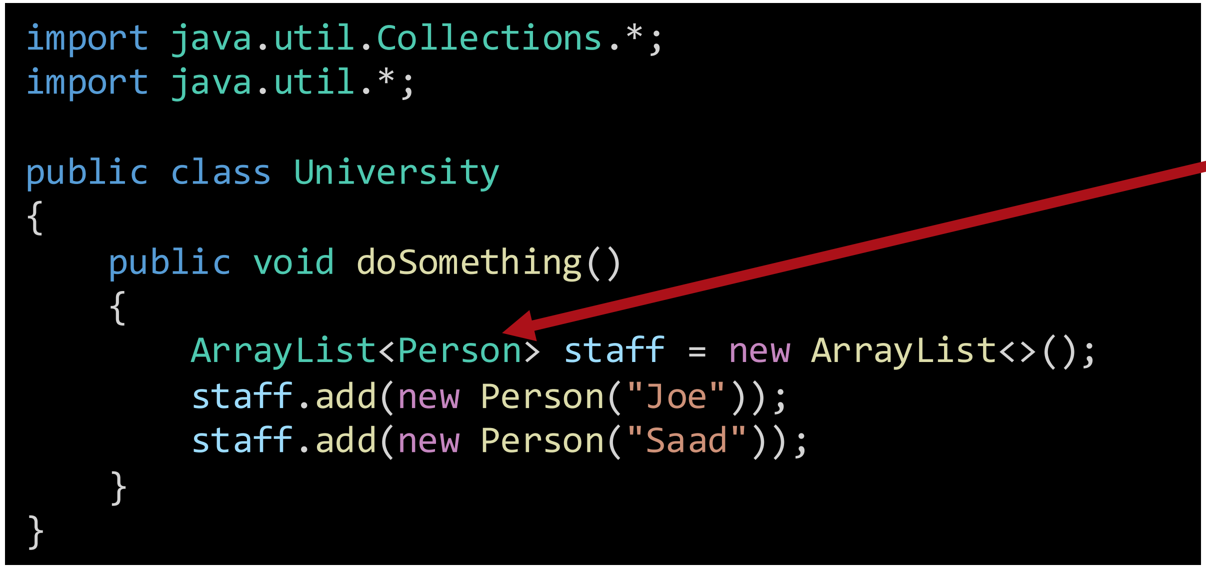
Some of these may be familiar from SCC121… ;)

ArrayList	**ordered, indexed list**
LinkedList	**ordered, non-indexed**
PriorityQueue	**FIFO with optional prioritisation**
HashSet	**unique, unordered**
TreeSet	**unique, ordered**
HashMap	**hashed key-value pairs (no ordering)**
TreeMap	**hashed key-value pairs (ordered by key)**

Using the Collections API

Collections are classes, so we just treat them as such.

- Create an object using its constructor
- Use its methods to interact with that data structure.
- Choose the best data structure for your application. **ArrayList** is a good default…



```
import java.util.Collections.*;
import java.util.*;

public class University
{
    public void doSomething()
    {
        ArrayList<Person> staff = new ArrayList<>();
        staff.add(new Person("Joe"));
        staff.add(new Person("Saad"));
    }
}
```

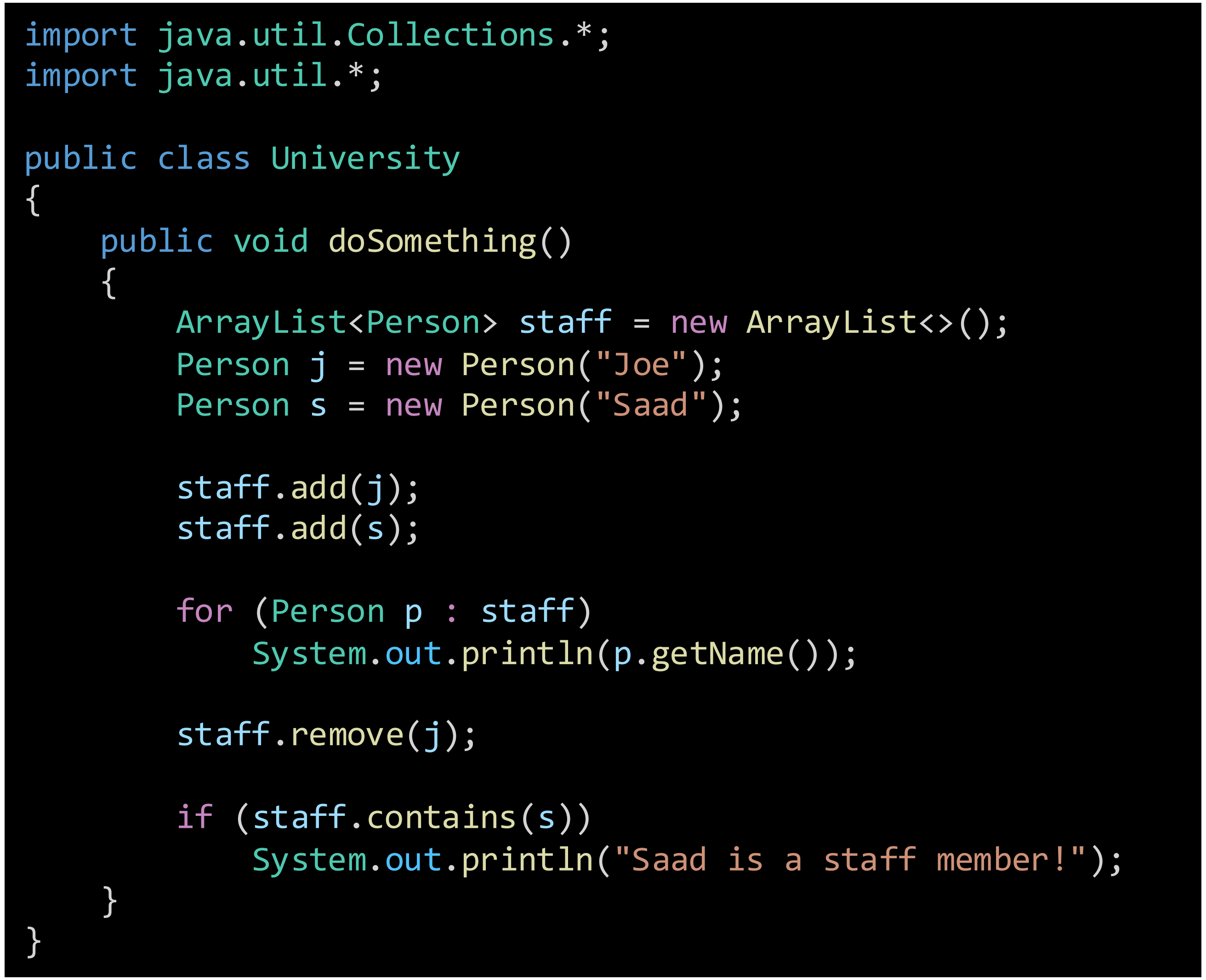
The type of the things you want to store

Using the Collections API…

We can use any of the methods defined in the relevant interfaces

- add
- remove
- contains
- get

We can also iterate over them in loops!



```
import java.util.Collections.*;
import java.util.*;

public class University
{
    public void doSomething()
    {
        ArrayList<Person> staff = new ArrayList<>();
        Person j = new Person("Joe");
        Person s = new Person("Saad");

        staff.add(j);
        staff.add(s);

        for (Person p : staff)
            System.out.println(p.getName());

        staff.remove(j);

        if (staff.contains(s))
            System.out.println("Saad is a staff member!");
    }
}
```

ArrayList

Acts much like an extensible array

- Items are maintained in a sequence, add/removed dynamically, etc.
- Items are also enumerated and indexed by location.

- Implemented internally as a simple array…
- if the array becomes full, a new one is created and the data copied from the old one.

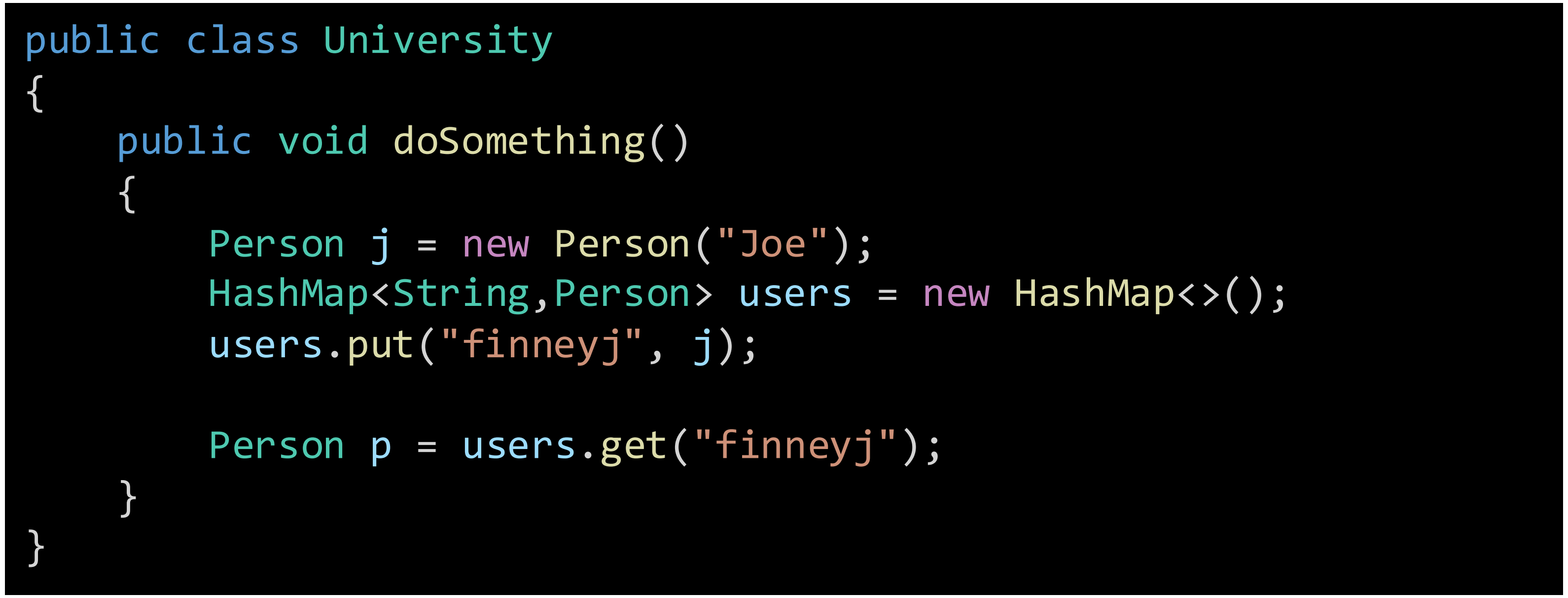
- **O(1)** complexity for index lookups
- **O(1)** complexity for additions (on average!)
- **O(n)** complexity for remove

- **Makes this a good choice for a general purpose data structure!**

HashMap

Unordered collection of key/value pairs

- Note here we define two types when creating an object (key and value)
- put() and get() methods allow us to add/remove objects from the collection
- **VERY fast. Approaching O(1) for key based addition, deletion, lookup...**

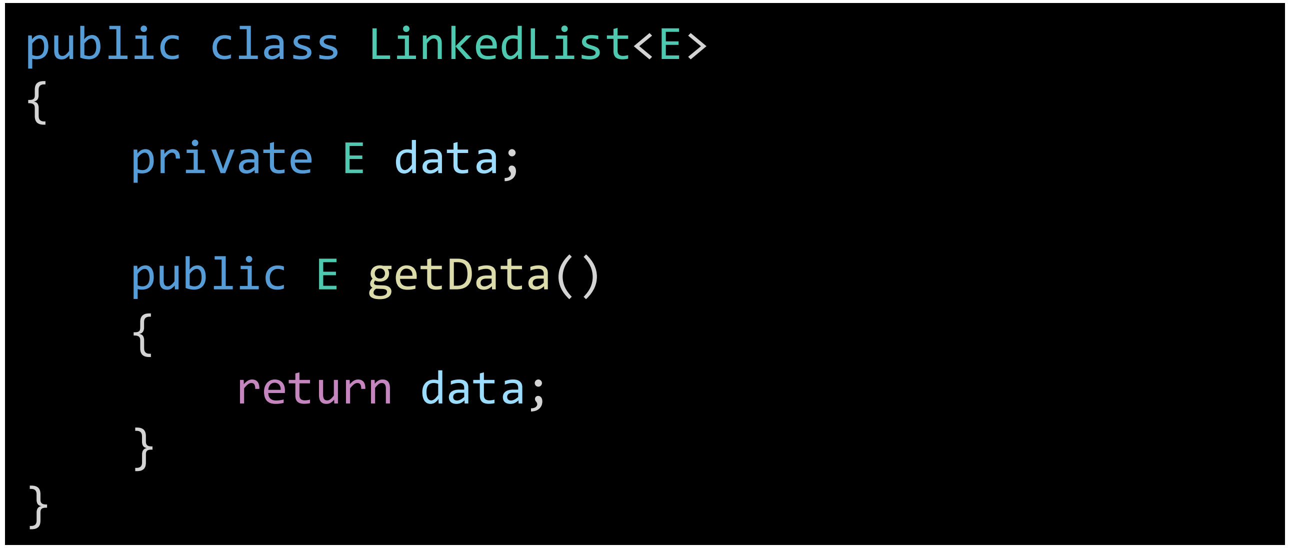


```
public class University
{
    public void doSomething()
    {
        Person j = new Person("Joe");
        HashMap<String,Person> users = new HashMap<>();
        users.put("finneyj", j);

        Person p = users.get("finneyj");
    }
}
```

Generics

Classes can be parameterised, just like functions and methods!

- Class definition can be appended with one or more **formal type parameters** in angled brackets.
- These parameters represent types that need to be defined when an object of that class can be created using new…
- Within the class, the formal type parameter can be used in instance variables, method signatures…

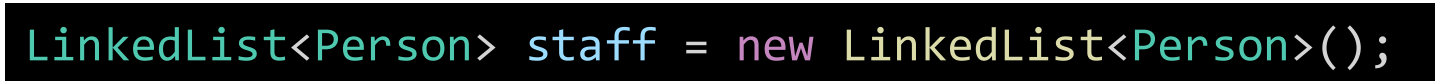


```java
public class LinkedList<E>
{
    private E data;

    public E getData()
    {
        return data;
    }
}
```

Using Generics

Formal type parameters bind to real types when objects are created using new

- The actual types are defined also in angled brackets at this point
- At this point, a new class is generated for that specific type...
- ...that class is then instantiated, and a strongly typed object reference returned.



```
LinkedList<Person> staff = new LinkedList<Person>();
```

Demo: A Simple Linked List

Summary

Today we introduced:

- Collections
- Generics
- Saw more examples of inheritance and polymorphism

Next Lecture:

- A surprise.
- ☺