SWEN20003
Object Oriented Software Development

Collections and Maps

Shanika Karunasekera
karus@unimelb.edu.au

University of Melbourne

The Road So Far

- Java Foundations
  - A Quick Tour of Java
- Object Oriented Programming Foundations
  - Classes and Objects
  - Arrays and Strings
  - Input and Output
  - *Software Tools and Bagel*
  - Inheritance and Polymorphism
  - Interfaces and Polymorphism
- Advanced Object Oriented Programming and Software Design
  - Modelling Classes and Relationships
  - Generics

Previous Lecture Generics

Learning Outcomes:

- Understand **generic** classes in Java
- Use **generically typed** classes
- Define **generically typed** classes

Previous Lecture Generics - Recap

We looked at how the type parameter `T was used in the Java Comparable Interface.`

```
public interface Comparable<T> {

    public int compareTo(T other);

}
```

```
public class Robot implements Comparable<Robot> {...}
public class Book implements Comparable<Book> {...}
public class Dog implements Comparable<Dog> {...}
```

Previous Lecture Generics - Recap

We looked at how to use the `ArrayList` class.

```
import java.util.ArrayList;
public class PrintCircleRadius {
    public static void main(String[] args) {
        ArrayList<Circle> circles = new ArrayList<Circle>();
        circles.add(new Circle(0.0, 0.0, 5));
        circles.add(new Circle(0.0, 0.0, 10));
        circles.add(new Circle(0.0, 0.0, 7));
        printRadius(circles);
    }
    private static void printRadius(ArrayList<Circle> circles){
        int index = 0;
        for(Circle c: circles) {
            System.out.println("Radius at index " + index +
            " = " +  c.getRadius());
            index++;
        }
    }
}
```

Lecture Objectives

After this lecture you will be able to:

- Choose appropriate data structures storing, retrieving and manipulating objects (data)
- Use the Java Collections Framework
- Use the Java Maps Framework

Collections and Maps

Understanding how to store data (a collection of objects) for later retrieval and manipulation is an essential when writing programs.

Collections and Maps

Understanding how to store data (a collection of objects) for later retrieval and manipulation is an essential when writing programs.

Java provides two *frameworks* to support this.

Collections and Maps

Understanding how to store data (a collection of objects) for later retrieval and manipulation is an essential when writing programs.

Java provides two *frameworks* to support this.

Keyword

Collections: A framework that permits storing, accessing and manipulating *lists* (an ordered collection).

Keyword

Maps: A framework that permits storing, accessing and manipulating *key-value pairs*.

Back to `ArrayList`

Last lecture we looked at the `ArrayList` as a generic class.

`ArrayList` is a class in the Java Collections framework that can be used for storing, retrieving and manipulating a group of objects.

In this lecture we will take a closer look at how we can use the `ArrayList` class for more sophisticated data manipulations.

Using the ArrayList Class for storing

```
import java.util.ArrayList;
public class PrintCircleRadius {
    public static void main(String[] args) {
        ArrayList<Circle> circles = new ArrayList<Circle>();
        circles.add(new Circle(0.0, 0.0, 5));
        circles.add(new Circle(0.0, 0.0, 10));
        circles.add(new Circle(0.0, 0.0, 7));
        printRadius(circles);
    }
    private static void printRadius(ArrayList<Circle> circles){
        int index = 0;
        for(Circle c: circles) {
            System.out.println("Radius at index " + index +
                               " = " +  c.getRadius());
            index++;
        }
    }
}
```

Using the `ArrayList` Class for storing

What would the program print?

```
Radius of circle: at index 0 = 5.0
Radius of circle: at index 1 = 10.0
Radius of circle: at index 2 = 7.0
```

Using the `ArrayList` Class for storing

`ArrayList` can be used for storing different types of objects, provided they inherit the same base class - therefore not quite different types of objects theoretically.

Why is this useful?

Common behaviour across objects can be executed seamlessly - see next example.

Using the ArrayList Class for storing

```
public abstract class Shape {
    public abstract double getArea();
}
public class Circle extends Shape {
    private double radius = 0.0;
    // Code for constructors, getter and setter go here
    @Override
    public double getArea() {
        return Math.PI*radius*radius;
    }
}
public class Square extends Shape {
    private double length = 0.0;
    // Code for constructors, getter and setter go here
    @Override
    public double getArea() {
        return length*length;
    }
}
```

Using the ArrayList Class for storing

```
import java.util.ArrayList;

public class ComputeAreaShapes {
    public static void main(String[] args) {
        ArrayList<Shape> shapes = new ArrayList<Shape>();
        shapes.add(new Circle(0.0, 0.0, 5));
        shapes.add(new Circle(0.0, 0.0, 10));
        shapes.add(new Square(0.0, 0.0, 7));
        printArea(shapes);
    }

    private static void printArea(ArrayList<Shape> shapes) {
        int index = 0;
        for(Shape s: shapes) {
            System.out.println("Area of shape: at index " +
                               index++ + " = " + s.getArea());
        }
    }
}
```

Using the `ArrayList` Class for storing

What would the program print?

```
Area of shape: at index 0 = 78.53981633974483
Area of shape: at index 1 = 314.1592653589793
Area of shape: at index 2 = 49.0
```

Using the `ArrayList` Class for sorting

Elements of an `ArrayList` can be easily sorted if:

Using the `ArrayList` Class for sorting

Elements of an `ArrayList` can be easily sorted if:

The stored element class implements the `Comaparable<T>` interface!

The `compareTo()` method of the class must provide a comparison (returning an integer) which will be used to decide how the elements are sorted.

Using the ArrayList Class for sorting

```
public class CircleT  implements Comparable<CircleT>{
    private double radius = 0.0;

    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public CircleT(double centreX, double centreY, int radius) {
        this.radius = radius;
    }

    @Override
    public int compareTo(CircleT c) {
        if (radius > c.getRadius())
            return 1;
        else if (radius < c.getRadius())
            return -1;
        else
            return 0;
    }
}
```

Using the ArrayList Class for sorting

```
import java.util.ArrayList;
import java.util.Collections;

public class PrintCircleRadiusSorted {
    public static void main(String[] args) {
        ArrayList<CircleT> circles = new ArrayList<CircleT>();
        circles.add(new CircleT(0.0, 0.0, 5));
        circles.add(new CircleT(0.0, 0.0, 10));
        circles.add(new CircleT(0.0, 0.0, 7));
        printRadius(circles);
        Collections.sort(circles);
        System.out.println("*****************************");
        printRadius(circles);
    }

    private static void printRadius(ArrayList<CircleT> circles){
        int index = 0;
        for(CircleT c: circles) {
            System.out.println("Radius of circle: at index " +
                        index++ + " = " + c.getRadius());
        }
    }
}
```

Using the ArrayList Class for sorting

What would the program print?

```
Radius of circle: at index 0 = 5.0
Radius of circle: at index 1 = 10.0
Radius of circle: at index 2 = 7.0
**************************
Radius of circle: at index 0 = 5.0
Radius of circle: at index 1 = 7.0
Radius of circle: at index 2 = 10.0
```

Using the ArrayList Class for sorting

```
import java.util.*;
class Movie implements Comparable<Movie>
{
    private double rating;
    private String name;
    private int year;
    public Movie(String name, double rating, int year)
    {
        this.name = name;
        this.rating = rating;
        this.year = year;
    }
    public int compareTo(Movie m)
    {
        return this.year - m.year;
    }
    // Getters and setters go here - not shown
}
```

Using the ArrayList Class for sorting

```
import java.util.ArrayList;
import java.util.Collections;
public class MovieSorter {
    public static void main(String[] args) {
        ArrayList<Movie> list = new ArrayList<Movie>();
        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));
        list.add(new Movie("Return of the Jedi", 8.4, 1983));
        Collections.sort(list);
        printList(list);
    }

    public static void printList(ArrayList<Movie> list) {
        for (Movie movie: list)
            System.out.println(movie.getRating() + " " +
                movie.getName() + " " +    movie.getYear());
    }
}
```

Using the `ArrayList` Class for sorting

What would the program print?

Using the `ArrayList` Class for sorting

What would the program print?

```
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980
8.4 Return of the Jedi 1983
8.3 Force Awakens 2015
```

Now, what if we want to sort the movies by rating or name - not year?

How can we do that?

Using the ArrayList Class for sorting

What would the program print?

```
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980
8.4 Return of the Jedi 1983
8.3 Force Awakens 2015
```

Now, what if we want to sort the movies by rating or name - not year?

How can we do that?

Good news is java `Comparator` and `Collections.sort()` can still help you!

Using the ArrayList Class for sorting

```
import java.util.Comparator;
class RatingComparator implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        if (m1.getRating() < m2.getRating()) return -1;
        if (m1.getRating() > m2.getRating()) return 1;
        else return 0;
    }
}

import java.util.Comparator;
public class NameComparator implements Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {
        return m1.getName().compareTo(m2.getName());
    }
}
```

Using the ArrayList Class for sorting

```
// import statements
public class MovieSorter {
    public static void main(String[] args) {
        // Code to add movies to the arraylist - same as pervious example
        Collections.sort(list);
        printList(list);
        System.out.println("*****************************");
        Collections.sort(list,new RatingComparator());
        printList(list);
        System.out.println("*****************************");
        Collections.sort(list,new NameComparator());
        printList(list);
    }
    public static void printList(ArrayList<Movie> list) {
        for (Movie movie: list)
            System.out.println(movie.getRating() + " " +
                movie.getName() + " " +    movie.getYear());
    }
}
```

Using the `ArrayList` Class for sorting

What would the program print?

Using the ArrayList Class for sorting

What would the program print?

```
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980
8.4 Return of the Jedi 1983
8.3 Force Awakens 2015
**********************************
8.3 Force Awakens 2015
8.4 Return of the Jedi 1983
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980
**********************************
8.8 Empire Strikes Back 1980
8.3 Force Awakens 2015
8.4 Return of the Jedi 1983
8.7 Star Wars 1977
```

Using the `ArrayList` Class for sorting

In the previous example, we developed new comparator class for each comparison.

Was it necessary? Is that a bit of an overkill?

Is there a different solution?

Using the `ArrayList` Class for sorting

In the previous example, we developed new comparator class for each comparison.

Was it necessary? Is that a bit of an overkill?

Is there a different solution?

Anonymous Inner Class is the solution.

Keyword

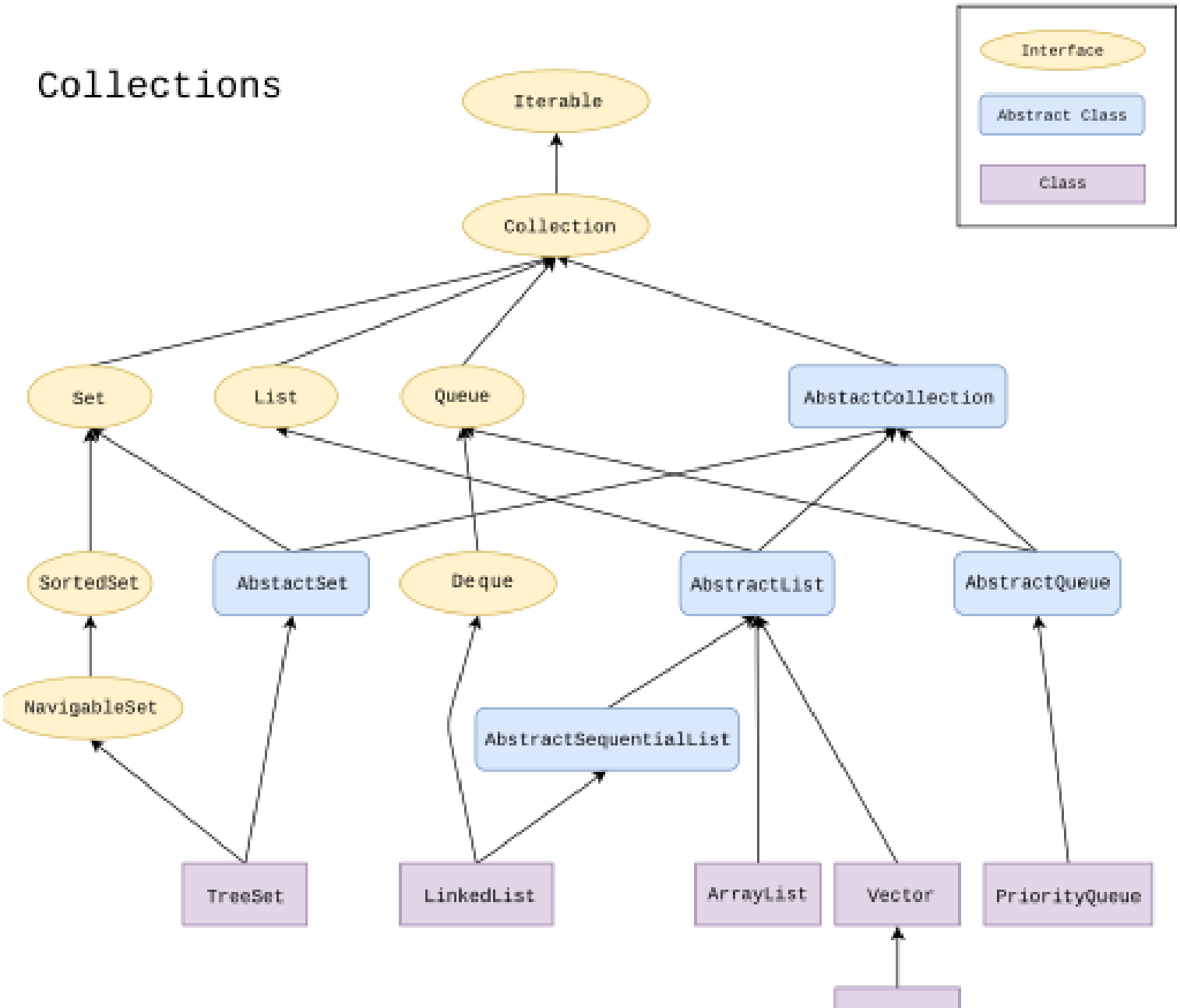
Anonymous Inner Class: A class created "on the fly", without a new file, or class name for which only a single object is created.

Using the ArrayList Class for sorting

```
public class MovieSorterAnnonymous {
    public static void main(String[] args) {
    // Same code as the previous example
        Collections.sort(list, new Comparator<Movie>(){
            @Override
            public int compare(Movie m1, Movie m2){
                if (m1.getRating() < m2.getRating()) return -1;
                if (m1.getRating() > m2.getRating()) return 1;
                else return 0;
            }});
        printList(list);

        Collections.sort(list, new Comparator<Movie>(){
            @Override
            public int compare(Movie m1, Movie m2) {
                return m1.getName().compareTo(m2.getName());
            }});
        printList(list);
    }
}
```

Collections Hierarchy



Common Operations - Collections

Length `int size()`

Common Operations - Collections

Length `int size()`

Presence `boolean contains(Object element)`

Only works when `element` defines `equals(Object element)`

Common Operations - Collections

Length	`int size()`
Presence	`boolean contains(Object element)` Only works when `element` defines `equals(Object element)`
Add	`boolean add(E element)`

Common Operations - Collections

Length	`int size()`
Presence	`boolean contains(Object element)` Only works when `element` defines `equals(Object element)`
Add	`boolean add(E element)`
Remove	`boolean remove(Object element)`

Common Operations - Collections

Length	`int size()`
Presence	`boolean contains(Object element)` Only works when `element` defines `equals(Object element)`
Add	`boolean add(E element)`
Remove	`boolean remove(Object element)`
Iterating	`Iterator<E> iterator()`

Common Operations - Collections

Length	`int size()`
Presence	`boolean contains(Object element)` Only works when `element` defines `equals(Object element)`
Add	`boolean add(E element)`
Remove	`boolean remove(Object element)`
Iterating	`Iterator<E> iterator()`
Iterating	`for (T t : Collection<T>)`
Retrieval	`Object get(int index)` Supported only at `AbstractList` level and below.

Most Useful?

Each of these have their useful applications, but personally...

- `ArrayList`: like arrays, but better

Most Useful?

Each of these have their useful applications, but personally...

- `ArrayList`: like arrays, but better
- `HashSet`: ensures elements are unique - no duplicates
- `PriorityQueue`: allows you to *order* elements in non-trivial ways

Most Useful?

Each of these have their useful applications, but personally...

- `ArrayList`: like arrays, but better
- `HashSet`: ensures elements are unique - no duplicates
- `PriorityQueue`: allows you to *order* elements in non-trivial ways
- `TreeSet`: Fast lookup/search of unique elements

Most Useful?

Each of these have their useful applications, but personally...

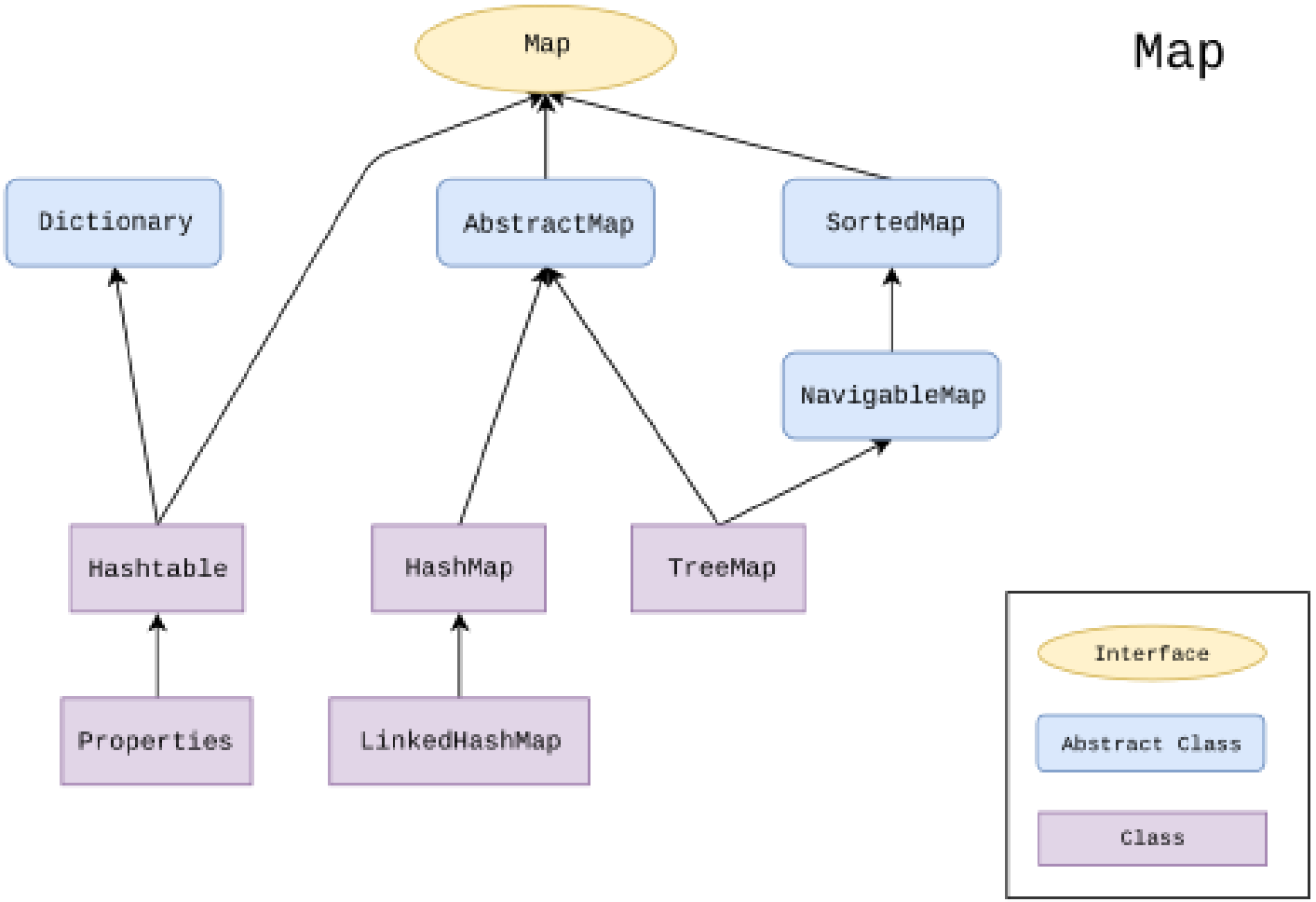
- `ArrayList`: like arrays, but better
- `HashSet`: ensures elements are unique - no duplicates
- `PriorityQueue`: allows you to *order* elements in non-trivial ways
- `TreeSet`: Fast lookup/search of unique elements

Maps

Keyword

Maps: A framework that permits storing, accessing and manipulating *key-value pairs*.

Maps Hierarchy



Source: https://en.wikipedia.org/wiki/Java_collections_framework [Note: Not UML]

Common Operations - Maps

Length `int size()`

Common Operations - Maps

Length	`int size()`
Presence	`boolean containKey(Object key)`
Presence	`boolean containValue(Object value)`
Add/Replace	`boolean put(K key, V value)`

Common Operations - Maps

Length	`int size()`
Presence	`boolean containKey(Object key)`
Presence	`boolean containValue(Object value)`
Add/Replace	`boolean put(K key, V value)`
Remove	`boolean remove(Object key)`

Common Operations - Maps

Length	`int size()`
Presence	`boolean containKey(Object key)`
Presence	`boolean containValue(Object value)`
Add/Replace	`boolean put(K key, V value)`
Remove	`boolean remove(Object key)`
Iterating	`Set<K> keySet()`

Common Operations - Maps

Length	`int size()`
Presence	`boolean containKey(Object key)`
Presence	`boolean containValue(Object value)`
Add/Replace	`boolean put(K key, V value)`
Remove	`boolean remove(Object key)`
Iterating	`Set<K> keySet()`
Iterating	`Set<Map.Entry<K,V>> entrySet()`
Retrieval	`V get(Object key)`

Using HashMap

A generic class that takes two types: `K` (the key) and `V` (the value)

```
import java.util.HashMap;

public static void main(String[] args) {
    HashMap<String,Book> library = new HashMap<>();

    Book b1 = new Book("J.R.R. Tolkien", "The Lord of the Rings", 1178);
    Book b2 = new Book("George R. R. Martin", "A Game of Thrones", 694);

    library.put(b1.author, b1);
    library.put(b2.author, b2);

    for(String author : library.keySet()) {
        Book b = library.get(author);
        System.out.format("%s, %s, %d\n", b.getAuthor(),
            b.getTitle(), b.getNumPages());
    }
}
```

Assess Yourself

If you were to create a digital phonebook using a `HashMap`, what would the `key` and `value` types be?

Assess Yourself

If you were to create a digital phonebook using a `HashMap`, what would the `key` and `value` types be?

```
HashMap<String,Integer> phonebook = new HashMap<>();
```

Assess Yourself

If you were to create a system to link a pet's ID to it's owner, what would the `key` and `value` types be?

Assess Yourself

If you were to create a system to link a pet's ID to it's owner, what would the `key` and `value` types be?

```
HashMap<Integer,Person> petTracker = new HashMap<>();
```

Assess Yourself

Write a class called `Tracker`, which accepts two type parameters. The first type must be subclass of `Person`, and the second type a subclass of `Locator`.

A `Person` object could be a `Hiker`, `Diver`, or `Pilot`.

A `Locator` object could be `GPS`, `Infrared`, or `IP`.

The `Tracker` class maintains a list of `TwoTypePair` objects, with the elements of the `TwoTypePair` being a `Person` and a `Locator`.

Generics in the Collections and Maps

If we didn't have generic classes, how would you implement a list, a map, etc.?

Generics in the Collections and Maps

If we didn't have generic classes, how would you implement a list, a map, etc.?

- Define everything as `Object`
- Rewrite your code for any type you might use it with

Generics in the Collections and Maps

If we didn't have generic classes, how would you implement a list, a map, etc.?

- Define everything as `Object`
- Rewrite your code for any type you might use it with

Generics give us **flexibility**; code once, reuse the code for **any** type. They also allow objects to keep their **type** (i.e. not be `Objects`), **and**, allows the compiler to detect errors, thereby prevent run-time errors if code is properly designed.

Lecture Objectives

After this lecture you will be able to:

- Choose appropriate data structures storing, retrieving and manipulating objects (data)
- Use Java Collections Framework
- Use Java Maps Framework