SWEN20003
Object Oriented Software Development

Generics II

Semester 1, 2019

The Road So Far

- Java Foundations
- Classes and Objects
    - Encapsulation
    - Information Hiding (Privacy)
- Inheritance and Polymorphism
    - Inheritance
    - Polymorphism
    - Abstract Classes
    - Interfaces
- Modelling classes and relationships
- Generics I

Previous Lecture Generics I - Recap

Learning Outcomes:

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

Previous Lecture Generics I - 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 I - Recap

We looked at how to use the ArryList class.

```java
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));
        Collections.sort(circles);
        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());
        }
    }
}
```

Lecture Objectives

After this lecture you will be able to:

- Develop your own **generic classes**
- Decide when generic programming is **appropriate**
- Use generic collection and map classes

Defining a Generic Class

Keyword

Generic Class: A class that is defined with an arbitrary type for a field, parameter or return type.

- The type parameter is included in angular brackets after the class name in the class definition heading.
- A type parameter can have any reference type (i.e., any class type) plugged in.
- Traditionally, a single uppercase letter is used for a type parameter, but any non-keyword identifier may be used.
- A class definition with a type parameter is stored in a file and compiled just like any other class.

Defining Generics

```java
public class Sample<T> {

    private T data;

    public void setData(T data) {
        this.data = data;
    }

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

Defining a Generic Class - Multiple Types

```java
public class TwoTypePair<T1, T2> {
    private T1 first;
    private T2 second;

    public TwoTypePair() {
        first = null;
        second = null;
    }

    public TwoTypePair(T1 first, T2 second) {
        this.first = first;
        this.second = second;
    }

    public void setFirst(T1 first){
        this.first = first;
    }

    public void setSecond(T2 second) {
        this.second = second;
    }
    // Additional methods go here
}
```

Using a Generic Class - Multiple Types

```java
import java.util.Scanner;
public class TwoTypePairDemo {
    public static void main(String[] args) {

        TwoTypePair<String, Integer> rating =
         new TwoTypePair<String, Integer>("The Car Guys", 8);

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Our current rating for " +
          rating.getFirst() + " is " + rating.getSecond());
        System.out.println("How would you rate them?");

        int score = keyboard.nextInt();
        rating.setSecond(score);
        System.out.println("Our new rating for "+
          rating.getFirst() + " is " + rating.getSecond());
    }
}
```

Bounded Type Parameters

Sometimes we need to *guarantee* a class' behaviour, so we apply *bounds* to type parameters.

```
public class Generic<T extends <class, interface...>> {
}
```

```
public class Generic<T extends Comparable<T>> {
}
```

```
public class Generic<T extends Robot> {
}
```

```
public class Generic<T extends Robot
                        & Comparable<T> & List<T>> {
}
```

Generic Methods

Keyword

Generic Method: A method that accepts arguments, or returns objects, of an arbitrary type.

A generic method can be defined in any class. The type parameter (e.g. `T`) is *local* to the method.

```java
public <T> int genericMethod(T arg); // Generic argument

public <T> T genericMethod(String name); // Generic return value

public <T> T genericMethod(T arg); // Both!

public <T,S> T genericMethod(S arg); // Both!
```

Assess Yourself

Write a generic method that accepts two arguments:

- `array`: an array of unknown type
- `item`: an object of the same type as the array

The method should return a count of how many times `item` appears in `array`.

Assess Yourself

```java
public class TestGenericMethods {
    public static void main(String[] args) {
        Integer[] nums = {1, 3, 6, 9, 3, 5, 9, 3, 5, 42, null};
        String[] names = {"Jon", "Arya", "Dany", "Tyrion", "Jon"};
        System.out.println(countOccurrences(nums, 3));
        System.out.println(countOccurrences(names, "Jon"));
    }

    public static <T> int countOccurrences(T[] array, T item) {
        int count = 0;
        if (item == null) {
            for (T arrayItem : array){
                count = arrayItem == item ? count + 1 : count;
            }
        } else {
            for (T arrayItem : array){
                count = item.equals(arrayItem) ? count + 1 : count;
            }
        }
        return count;
    }
}
```

Pitfall: What Can't We Do?

Generic programming is powerful, but has its limitations. When using generics, we can't:

- Instantiate parametrized objects

```
T item = new T();
```

- Create arrays of parametrized objects

```
T[] elements = new T[];
```

Otherwise, most things are fair game.

Lecture Objectives

After this lecture you will be able to:

- Develop your own **generic classes**
- Decide when generic programming is **appropriate**
- *Use Java generic collection and map classes*

Back to ArrayList and sorting

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

We also looked at how items in the list could be sorted using Collections.sort() method, provided that:

the class to be sorted implemented the java Comparable interface.

Back to ArrayList and sorting

```java
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
}
```

Back to `ArrayList` and sorting

```java
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());
    }
}
```

Back to ArrayList and 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!

Back to ArrayList and 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());
    }
}
```

Back to ArrayList and 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());
    }
}
```

Back to ArrayList and 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
```

Back to `ArrayList` and sorting

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.

Back to ArrayList and sorting

```java
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);
    }
}
```

Is That It?

Is `ArrayList` the only option for generic behaviour?

Of course not! Java has more *frameworks*...

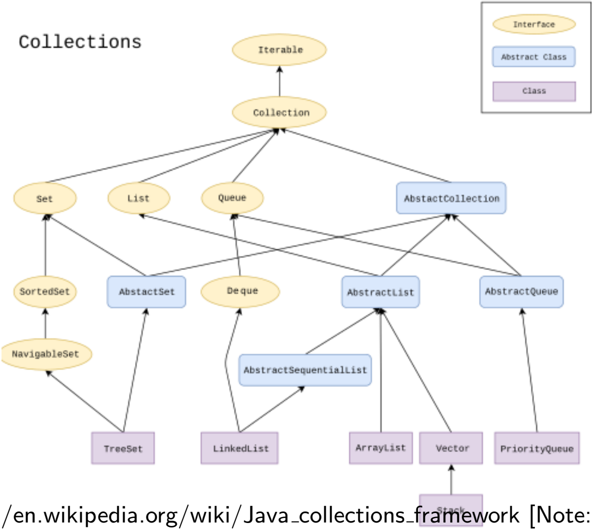
Keyword

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*.

Collections Hierarchy



Source: https://en.wikipedia.org/wiki/Java_collections_framework [Note: Non-UML Notation]

Common Operations - Collections

Length	`int size()`

Presence	`boolean contains(Object element)`
	Only works when `element` defines
	`equals(Object element)`

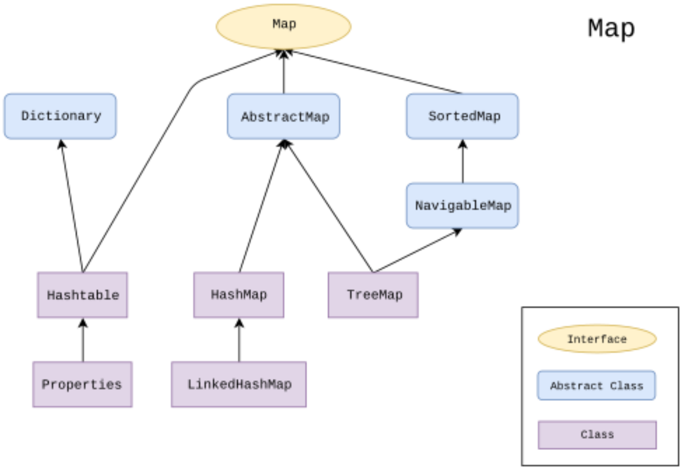
Add	`boolean add(E element)`
Remove	`boolean remove(Object element)`
Iterating	`Iterator<E> iterator()`
Iterating	`for (T t : Collection<T>)`
Retrieval	`Object get(int index)` Supported only at `AbstractList` level and below.

Most Useful?

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

- `ArrayList`: like arrays, but better
- `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 Hierarchy



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

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?

```
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?

```
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.

Why Generics?

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

You should be able to:

- Develop your own **generic classes**
- Decide when generic programming is **appropriate**
- Use Java generic collection and map classes