SWEN20003
Object Oriented Software Development

Generics I

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

Recap - Modelling classes and relationships

Learning Objectives:

- Identify classes and **relationships**
- Represent classes and relationships in **UML**
- Develop simple **Class Diagrams**

Lecture Objectives

After this lecture you should be able to:

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

Introduction

Java allows class, interface or method definitions to include **parameter types**.

Such definitions are called generics:

- Enables generic logic to be written that applies to any class type
- Allows code re-use

Example: What about a generic `Sort` class that would allow any type of object to be sorted?

A look back...

You have already seen a generic interface.

What is the Comparable interface? How does it work?

```
public interface Comparable<T> {

    public int compareTo(T other);

}
```

What does `T` mean?

Type Parameters

- `T` is a *type parameter*, or type variable
- The value of `T` is literally a type (class/interface); `Integer`, `String`, `Robot`, `Book`, `Driveable`
- When `T` is given a value (type), every instance of the *placeholder* variable is replaced

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

Type Parameters

How do you write a class that can be compared with an object of the same type?

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

    private String name;

    public Dog(String name) {
        this.name = name;
    }

    public int compareTo(Dog dog) {
        return this.name.compareTo(dog.name);
    }

}
```

Type Parameters

Using type parameters allows us to define a class or method that uses arbitrary, **generic** types, that applies to **any** and **all** types.

But why?

Can we compare objects without using the generic `Comparable` interface?

Using the Non-generic Comparable Intf.

```
public class Circle implements Comparable {
    private double centreX = 0.0,  centreY = 0.0;
    private double radius = 0.0;
    @Override
    public int compareTo(Object o) {
        Circle c = null;
        if (o instanceof Circle) {
            c = (Circle)o;
            if (c.radius > this.radius)
                return 1;
            else if (c.radius < this.radius)
                return -1;
            else
                return 0;
        } else {
            return -2;
        }
    }
}
```

Using the Non-generic Comparable Intf.

```
public class Square implements Comparable{
    private double centreX = 0.0, centreY = 0.0;
    private double length = 0.0;
    @Override
    public int compareTo(Object o) {
        Square s = null;
        if (o instanceof Square) {
            s = (Square)o;
            if (s.length > this.length)
                return 1;
            else if (s.length < s.length)
                return -1;
            else
                return 0;
        } else {
            return -2;
        }
    }
```

Using the Non-generic Comparable Intf.

```
public class CompareShapes {
    public static void main(String[] args) {
        Circle c1 = new Circle(0.0, 0.0, 5);
        Circle c2 = new Circle(0.0, 0.0, 10);
        System.out.println("Compare c1 and c2
                                = " + c1.compareTo(c2));
        Square s =  new Square(0.0, 0.0, 10);
        System.out.println("Compare c1 and s
                                = " + c1.compareTo(s));
    }
}
```

Yes it works, but the solution is not elegant!
The programmer has to check for -2 which is not a valid comparison.

Can we avoid this?

Using the Generic Comparable Intf.

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

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

Assume you also have a SquareT class which implements the generic Comparable interface.

Using the Generic Comparable Intf.

```
public class CompareShapesT {
    public static void main(String[] args) {
        CircleT c1 = new CircleT(0.0, 0.0, 5);
        CircleT c2 = new CircleT(0.0, 0.0, 10);
        System.out.println("Compare c1 and c2 = "
                           + c1.compareTo(c2));
        SquareT s =  new SquareT(0.0, 0.0, 10);
        System.out.println("Compare c1 and s = "
                           + c1.compareTo(s));
        //The line above will give a compiler error
    }
}
```

Using the `ArrayList` Class

What are the limitations of array?

- Finite length
- Resizing is a manual operation
- Requires effort to "add" or "remove" elements

Is there an alternative?

`ArrayList` class, which is a generic class, solves the above problems!

Using the ArrayList Class

```
import java.util.ArrayList;
public class PrintCircleRadius {
    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);
    }
    private static void printRadius(ArrayList<CircleT> circles){
        int index = 0;
        for(CircleT c: circles) {
            System.out.println("Radius at index " + index +
                                    " = " +  c.getRadius());
            index++;
        }
    }
}
```

Using the `ArrayList` Class

So what does the `ArrayList` give you?

- Can be iterated like arrays (for-each)
- Automatically handles resizing
- Can *insert*, *remove*, *get*, and *modify* elements at any index (plus many more capabilities)
- Inherently able to `toString()`
- Can't be **indexed** ([ ])

`ArrayList` is a class with an *array* as an instance variable.

Using the `ArrayList` Class

Are there any limitations of the `ArrayList` class?

- Although an ArrayList grows automatically when needed, it does not shrink automatically, hence can consume more memory than required - `trimToSize()` method must be invoked to release the excess memory.
- Cannot store primitive data types (int, float, etc.).

Using the `ArrayList` Class

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

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

Using the `ArrayList` Class

`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

```
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

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

Assess Yourself

Implement the method

`ArrayList<Integer> generateList(Scanner scanner)`

which continually accepts integers from a user until they end input, and returns an `ArrayList` of all the integers entered.

Implement a second method

`double average(ArrayList<Integer> numbers)`

that returns the average of all elements in `numbers`.

Assess Yourself

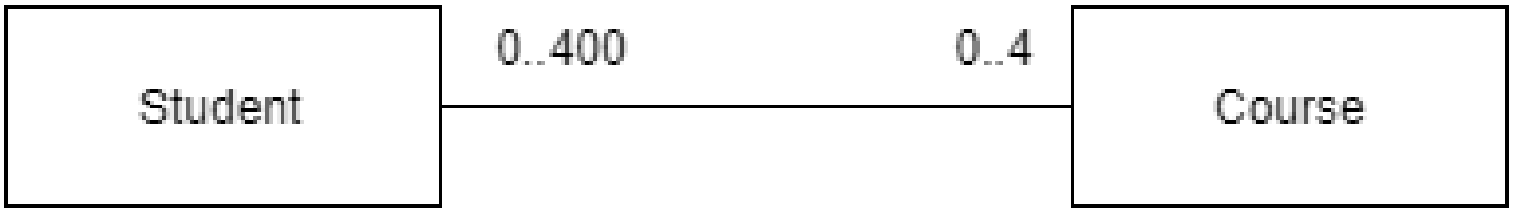
```java
public ArrayList<Integer> generateList(Scanner scanner) {

    ArrayList<Integer> numbers = new ArrayList<>();

    while (scanner.hasNextInt()) {
        numbers.add(scanner.nextInt());
    }

    return numbers;

}
```

Assess Yourself

```
public double average(ArrayList<Integer> numbers) {

    double sum = 0.0;

    for (Integer number : numbers) {
        sum += number;
    }

    return sum / numbers.size();

}
```

Assess Yourself

Now we can implement this relationship!



Implement the `Student` and `Course` classes, ignoring all instance variables but what the diagram shows.

Include methods to enrol a `Student` in a `Course`, and vice versa.

Assess Yourself

```
public class Student {

    public static final int MAX_COURSES = 4;

    private ArrayList<Course> courses = new ArrayList<>();

    public boolean addCourse(Course course) {
        if (courses.size() < MAX_COURSES &&
                !courses.contains(course)) {
            courses.add(course);
            course.addStudent(this);
            return true;
        } else {
            return false;
        }
    }

}
```

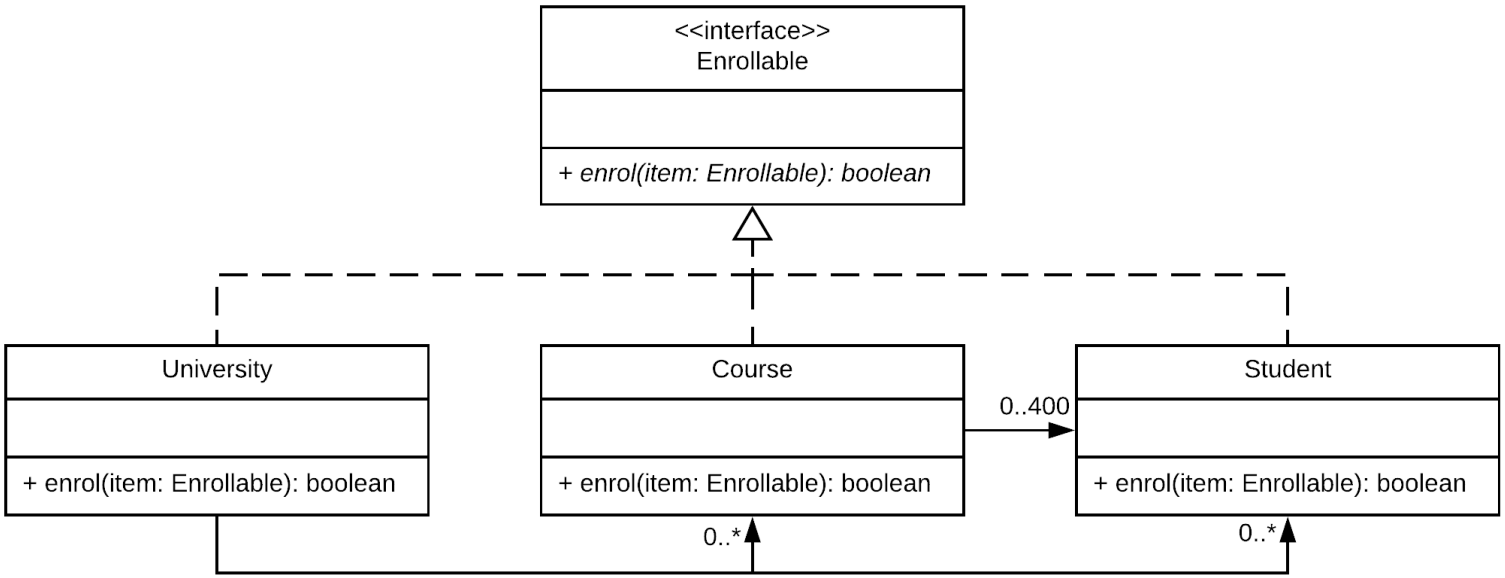
Assess Yourself

```
public class Course {

    public static final int MAX_STUDENTS = 400;

    private ArrayList<Student> students = new ArrayList<>();

    public boolean addStudent(Student student) {
        if (students.size() < MAX_STUDENTS &&
                !students.contains(student)) {
            students.add(student);
            return true;
        } else {
            return false;
        }
    }

}
```

Assess Yourself

Think carefully... Is there something odd about our design? An abstraction we've missed?

How could we make adding or changing features easier?



What does this look like?

Assess Yourself

```
import java.util.ArrayList;

public class Student implements Enrollable<Course> {
    public static final int MAX_COURSES = 5;

    ArrayList<Course> courses = new ArrayList<>();

    public boolean enrol(Course course) {
        if (courses.size() < MAX_COURSES &&
                !courses.contains(course)) {
            courses.add((Course)course);
            course.enrol(this);
            System.out.println("Added course successfully");
            return true;
        } else {
            System.out.println("Failed to add course");
            return false;
        }
    }

}
```

Assess Yourself

Write a main method that allows a user to continually enter integers, where each number is added to an `ArrayList`.

The program should then create a `PriorityQueue` for arranging the numbers based on the following rules:

- Ascending order
- Descending order
- Shortest (least characters) first
- Ascending order by the last (rightmost) digit

Learning Outcomes

You should be able to:

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