SWEN20003
Object Oriented Software Development

Generics

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

Lecture Objectives

After this lecture you should be able to:

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

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

We will first learn how to **use** generically typed classes and then learn how to **write** generically typed classes.

A look back...

Do you remember the sorting example in the Interfaces and Polymorphism lecture?

A look back...

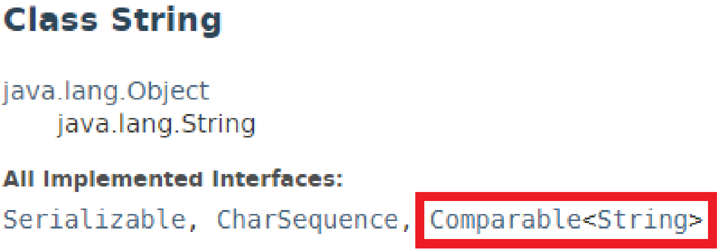
Do you remember the sorting example in the Interfaces and Polymorphism lecture?

The Comparable interface we used..

A look back...

Do you remember the sorting example in the Interfaces and Polymorphism lecture?

The Comparable interface we used..



Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

`Serializable, CharSequence, Comparable<String>`

Comparable Interface

How was the Comparable interface defined?

Comparable Interface

How was the Comparable interface defined?

```java
public interface Comparable<T> {

    public int compareTo(T other);

}
```

Comparable Interface

How was the Comparable interface defined?

```java
public interface Comparable<T> {

    public int compareTo(T other);

}
```

What does `T` mean?

Type Parameters

- `T` is a *type parameter*, or type variable

Type Parameters

- `T` is a *type parameter*, or type variable
- When `T` is given a value (type), every instance of the *placeholder* variable is replaced

Type Parameters

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

Type Parameters

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

```
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?

Type Parameters

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

```java
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.

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.

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

Using the Non-generic `Comparable` Intf.

What would the program print?

Using the Non-generic Comparable Intf.

What would the program print?

```
Compare c1 and c2 = 1
Compare c1 and s = -2
```

Using the Non-generic Comparable Intf.

What would the program print?

```
Compare c1 and c2 = 1
Compare c1 and s = -2
```

Yes it works, but the solution is not elegant!

Using the Non-generic `Comparable` Intf.

What would the program print?

```
Compare c1 and c2 = 1
Compare c1 and s = -2
```

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

Using the Non-generic `Comparable` Intf.

What would the program print?

```
Compare c1 and c2 = 1
Compare c1 and s = -2
```

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.

```java
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.

```java
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

We will next learn how to use `ArrayList`, a useful generics class that overcomes the limitations of arrays.

What are the limitations of array?

Using the ArrayList Class

We will next learn how to use ArrayList, a useful generics class that overcomes the limitations of arrays.

What are the limitations of array?

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

Using the ArrayList Class

We will next learn how to use `ArrayList`, a useful generics class that overcomes the limitations of arrays.

What are the limitations of array?

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

Using 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++;
        }
    }
}
```

Using the ArrayList Class

What would the program print?

Using the ArrayList Class

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

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?

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.).

We will learn more about the ArrayList class in our next topic on Collection and Maps - ArrayList is a class in the java Collections framework.

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...>> {
}
```

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>> {
}
```

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> {
}
```

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
```

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
```

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.

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

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

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

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.

```
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!
```

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.

Learning Outcomes

You should be able to:

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