OBJECT ORIENTED PROGRAMMING - LAB 9

7th November 2023

Learning Objectives:

- Generics
- Bounding Generic Objects
- Wildcards
- Interfaces (revision)
- Inheritance (revision)

Genrics

Generics simply means parameterising the code. One of the main ideas behind Object Oriented Paradigms is abstracting root-level details and creating as general as possible designs. We have already used inheritance and interfaces for this. But many a times, we want our code to behave differently for different data types. Consider the example:

```
class Data<T> {
  private T data;

  public Data(T data) {
    this.data = data;
  }

  public void printData() {
    System.out.println(data);
  }
}
```

In this example, we have a *Data* class which can store data of any type! Imagine we need Data classes of Integers, Strings and Arrays. Without generics, we would need one class for each data type. Here the code can be re-used. But generics must be used ***without loss of generality!*** This means that the code inside the Data class must be as general as possible and applicable for all data types. We will resolve the shortcomings of this using bounded generics.

Why Genrics?

You must be wondering - *“we already have an Object class that is the parent of every object. Why not write code that checks types using instanceOf and then type cast objects as and when required?”* This is very wrong due to type safety issues. A programmer may or may not check the instance of the objects and cast them directly leading to runtime errors. When building large applications, it is always desirable to catch issues at compile-time rather than waiting for something wrong to happen at run-time (when millions of users are using the app). Generics solves this problem by ensuring type-safety at compile time. Additionally, Generics largely remove the need for programmer-side type casting by introducing general code that works for everything! Hence, along with safety, it also brings in code re-usability and compatibility with various data types.

Bounded Generics

What if our Data class has some methods that are specific to Numbers (float, double, integer)? Now our class cannot store strings or arrays! We ensure this using bounded generics. Bounded Generics allows us to implement **restricted parameterisation** of classes. Here is the basic syntax:

```
class Data<T extends Number> {
   private T data;

   public Data(T data) {
     this.data = data;
   }

   public void increment() {
     this.data ++;
   }
}
```

Now our Data class will only accept data of classes that inherit Number class (Float, Integer etc). Note that the syntax is a bit confusing! *class A extends B* means that A **inherits** from class B, whereas *class A<T extends B>* means that the objects of type T must **inherit** B (if B is a class) **OR** must **implement** B (if B is an interface).

Wildcards for Bounding

In Java, wildcards are a feature of generics that allow you to write more flexible and reusable code when working with generic types. Wildcards are represented by the question mark **(?)** and are used in generic type declarations to indicate that a type parameter can be any unknown type. An example of **upper bound wildcard** is shown:

```
class Data<T extends Number> {
   private T data;
   ...
   public compare(Data<? extends Number>) {
      // code for comparison
}
```

In the compare method we place an *upper bound* that we can can compare with any Data whose data field inherits from Number. Do read about Lower Bounded and Unbounded wildcards after the lab.

How does Generics work?

Internally, many languages like Java implement Generics using **Type Erasure**. In simple terms, what happens is that the compiler replaces type in Data<T> with something concrete (like Object). Then at compile-time, it performs **type-checking** based on the original generic code. It ensures that the operations and assignments within the generic code comply with the specified type parameters. Finally, when you retrieve values from generic classes or methods, the compiler inserts type casting to ensure that you get the expected type. This type casting is done at runtime and is called **unchecked casting**.

Lab Exercise Question

Cricket fever is on! This week's lab is on designing a system for an online sports streaming platform. **Read the instructions (in the comments)** about each class before implementing them (you can implement them in this order):

- **Player**
- **CricketPlayer**
- **FootballPlayer**
- **Team**
- **CricketTeam**
- **FootballTeam**
- **League**
  - addTeam()
  - playMatch()
  - updateRanking()
  - getWinners()
  - showStandings()
  - assignRandomScoresToTeam1()

NOTE: You will be requiring the following the **Random** Class from java utils. Here are some basic functionalities:

```
import java.util.Random;          // already done for you
....
....
....
Random rand = new Random();       // new random object
int score = rand.nextInt(100);    // generates a random number in the
                                     range (0,100] i.e. 0 inclusive
                                     100 exclusive
```

Submission Guidelines

- After completing the lab, **compress your solution in a .zip format.** The zip file and the folder inside it should have the name format **202XYYYY9999G_Lab9.**

- **Upload before completion time i.e. 3:50:00 PM.** To be safe, you can upload your solution near the completion time and continue working to avoid missing the deadline. **No excuse regarding late submission will be handled.**