BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
CS F213
LAB 5

AGENDA **TIME: 02 Hrs**

- Interfaces
- Nested Interfaces
- Nested Classes
- Generics

5.1 Java Interfaces

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body). We use the interface keyword to create an interface in Java. For example,

```
interface Language {
  public void getType();

  public void getVersion();
}
```

Here,

- Language is an interface.
- It includes abstract methods: getType() and getVersion().

Why do we use interfaces ?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

Internal addition by the compiler:

The Java compiler **adds public and abstract keywords before the interface method.** Moreover, it adds public, static and final keywords before data members.

Program 5.1.a: Java Interface

```
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
        System.out.println("The area of the rectangle is " + (length * breadth));
  }
}

class Main {
  public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.getArea(5, 6);
  }
}
```

Output:
The area of the rectangle is 30

In the above example, we have created an interface named Polygon. The interface contains an abstract method getArea(). Here, the Rectangle class implements Polygon. And, provides the implementation of the getArea() method.

Program 5.1.b:

```
// create an interface
interface Language {
  void getName(String name);
}
```

```
// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
        System.out.println("Programming Language: " + name);
  }
}

class Main {
  public static void main(String[] args) {
        ProgrammingLanguage language = new ProgrammingLanguage();
        language.getName("Java");
  }
}
```

Output:

```
Programming Language: Java
```

In the above example, we have created an interface named Language. The interface includes an abstract method getName(). Here, the ProgrammingLanguage class implements the interface and provides the implementation for the method.

5.1.2 Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces.

```
example:
interface A {
  // members of A
}

interface B {
  // members of B
}

class C implements A, B {
  // abstract members of A
  // abstract members of B}
```

Program 5.1.c:

```
interface Flyable {
	void fly();
}

interface Eatable {
	void eat();
}

// Bird class will implement both interfaces
class Bird implements Flyable, Eatable {

	public void fly() {
		System.out.println("Bird flying");
	}

	public void eat() {
		System.out.println("Bird eats");
	}
	// It can have more methods of its own.
}

/*
 * Test multiple interfaces example
 */
public class Sample {

	public static void main(String[] args) {

		Bird b = new Bird();
		b.eat();
		b.fly();
	}

}
```

5.1.3 Extending an Interface

Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

Program 5.1.d:

```
interface A {
   void funcA();
}
interface B extends A {
   void funcB();
}
class C implements B {
   public void funcA() {
      System.out.println("This is funcA");
   }
   public void funcB() {
      System.out.println("This is funcB");
   }
}
public class Demo {
   public static void main(String args[]) {
      C obj = new C();
      obj.funcA();
      obj.funcB();
   }
}
```

5.1.4 Extending Multiple Interfaces

An interface can extend multiple interfaces.

example:

```
interface A {
  ...
}
interface B {
  ...
}

interface C extends A, B {
  ...
}
```

Program 5.1.e

```
interface A {
  public void test();
  public void test1();
}
interface B {
  public void test();
  public void test2();
}
interface C extends A,B {
  public void test3();
}
class D implements C {
  public void test() {
    System.out.println("Testing\n");
  }
  public void test1() {
    System.out.println("Testing1\n");
  }
  public void test2() {
    System.out.println("Testing2\n");
  }
  public void test3() {
    System.out.println("Testing3");
  }
}
public class Main {
```

```
  public static void main(String[] args) {
    D d=new D();
    d.test();
    d.test1();
    d.test2();
    d.test3();
  }
}
```

5.1.5 Default methods in Java Interfaces

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.

To declare default methods inside interfaces, we use the default keyword. For example,

```
public default void getSides() {
  // body of getSides()
}
```

Program 5.1.f: Default Method in Java Interface

```
interface Polygon {
  void getArea();

  // default method
  default void getSides() {
    System.out.println("I can get sides of a polygon.");
  }
}

// implements the interface
class Rectangle implements Polygon {
  public void getArea() {
    int length = 6;
    int breadth = 5;
    int area = length * breadth;
    System.out.println("The area of the rectangle is " + area);
  }

  // overrides the getSides()
```

```
  public void getSides() {
    System.out.println("I have 4 sides.");
  }
}

// implements the interface
class Square implements Polygon {
  public void getArea() {
    int length = 5;
    int area = length * length;
    System.out.println("The area of the square is " + area);
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of Rectangle
    Rectangle r1 = new Rectangle();
    r1.getArea();
    r1.getSides();

    // create an object of Square
    Square s1 = new Square();
    s1.getArea();
    s1.getSides();
  }
}
```

Output
The area of the rectangle is 30
I have 4 sides.
The area of the square is 25
I can get sides of a polygon.

5.1.6 Default Methods & Multiple Inheritance

Program 5.1.g

```
interface Printable{
```

```
 void print();
 default void show(){
    System.out.println("Within Printable");
 }
}
interface Showable{
 default void show(){
    System.out.println("Within Showable");
 }
void print();
}

class trial implements Printable, Showable{
 public void show(){
  Printable.super.show();
  Showable.super.show();
 }
 public void print(){
   System.out.println("Within Print"); }
 }
public class test{
 public static void main(String[] args){
  trial t = new trial();
  t.print();
  t.show();
 }
}
```

Output:

```
Within Print
Within Printable
Within Showable
```

5.1.7 Exercise

Question 1: What is wrong with the following interface?

```
public interface SomethingIsWrong {
        void aMethod(int aValue) {
        System.out.println("Hi Mom");
        }
}
```

Question 2: Fix the interface in Question 1.

Question 3: Is the following interface valid?

```
public interface Marker {
}
```

Question 4. Consider the following two classes:

```
public class ClassA {
   public void methodOne(int i) {
   }
   public void methodTwo(int i) {
   }
   public static void methodThree(int i) {
   }
   public static void methodFour(int i) {
   }
}

public class ClassB extends ClassA {
   public static void methodOne(int i) {
   }
   public void methodTwo(int i) {
   }
   public void methodThree(int i) {
   }
   public static void methodFour(int i) {
```

```
    }
}
```

a. Which method overrides a method in the superclass?
b. Which method hides a method in the superclass?
c. What do the other methods do?

Answers:

Answer 1: It has a method implementation in it. Only default and static methods have implementations.

Answer 2:

```
public interface SomethingIsWrong {
        void aMethod(int aValue);
}
```

Alternatively, you can define aMethod as a default method:

```
public interface SomethingIsWrong {
        default void aMethod(int aValue) {
        System.out.println("Hi Mom");
        }
}
```

Answer 3: Yes. Methods are not required. Empty interfaces can be used as types and to mark classes without requiring any particular method implementations. For an example of a useful empty interface, see java.io.Serializable.

Answer 4:
Answer 4a: methodTwo
Answer 4b: methodFour
Answer 4c: They cause compile-time errors.

5.2 Nested Interfaces

Nested or Member Interfaces are the interfaces declared as member of a class or another interface. These interfaces are declared to be static implicitly.

5.2.1 Within Classes

Usually, interfaces can only be declared with public or default access modifiers. But **nested interfaces within a class can have theoretically any access modifier**, however private access modifier doesn't have many use cases as it can't be accessed outside the nesting class.

Nested Interface in such a scenario is referenced (or implemented) as **nesting_class_name.nested_interface_name**.

Program 5.2.a

```
// Java program to demonstrate working of
// nested interface inside a class.
import java.util.*;
class NestingClass
{
   interface NestedInterface
   {
      void show();
   }
}

class Learning implements NestingClass.NestedInterface
{
   public void show()
   {
      System.out.println("show method of interface");
   }
}

public class A
{
   public static void main(String[] args)
   {
```

```
        ___X__  t = new Learning();
        t.show() ;
    }
}
```

Exercises

1. Fill the blank X in two different ways.
2. Try using different access modifiers for NestedInterface and note down the results.

// Answers of 1) are Learning and NestingClass.NestedInterface
// Answers of 2) are
private : compilation error ; private access
default or package private : show method of interface
protected : show method of interface
public : show method of interface

5.2.2 Within Interfaces

Nested interfaces within an interface can only have a public access modifier. Important point to note is that even if any access modifier is not specified for the nested interface, then also **it will have a public access modifier not default**. While implementing nested interfaces, there is no need to implement methods of nesting interface (interface in which nested interface is present).

Reason : It is because interface members can be only public.

Nested Interface in such a scenario is referenced (or implemented) as nesting_interface_name.nested_interface_name.

```
Program 5.2.b
// Java program to demonstrate working of
// nested interface inside a interface
import java.util.*;
interface NestingInterface
{
    interface NestedInterface
    {
        void show();
```

```
    }
void show1();
}

class Learning implements NestingInterface.NestedInterface
{
    public void show()
    {
        System.out.println("show method of interface");
    }
}

public class A
{
    public static void main(String[] args)
    {
        ___X__  t = new Learning();
        t.show() ;
    }
}
```

Exercises

1. Fill the blank X in two different ways.
2. Try using different access modifiers for NestedInterface and note down the results.
3. Verify whether we need to implement show1 necessarily in Learning class.

// Answers of 1) are Learning and NestingInterface.NestedInterface

// Answers of 2) are

private : compilation error ; error: illegal combination of modifiers.

default or package private : Implicitly nested interface is public in this case.

protected : compilation error ; error: illegal combination of modifiers.

public : show method of interface

Questions to think about

- The reason for the “ illegal combination of modifiers ” error?
- Why would nested interfaces be declared static implicitly, what’s the logic?

- What is the general idea behind allowing nested interfaces?

  // hint :- grouping related interfaces for maintenance purposes.
  // example of nested interfaces from Java Collection (to be covered later) is Entry, which is the subinterface of Map, i.e., accessed by Map.Entry.

Additional Important Scenarios

1. Define a public interface named Map, with an integer variable named 'max_elements' initialized to 10, a nested interface named Entry and a member function named func. Nested Interface Entry has an integer variable named 'max_elements' initialised to 5, and a member function named show. (Don't add any other methods / variables).

   a. Write a class implementing Map interface, also having an integer variable 'max_elements' initialized to 7. Determine which function needs to be necessarily implemented in this class and why, and how to print Map's max_element inside any of the functions implemented in this class.

      Your Ans:

   b. Similarly, write a class implementing Entry interface, also having an integer variable 'max_elements' initialized to 8. Determine which function needs to be necessarily implemented in this class, and how to print Entry's max_element inside any of the functions implemented in this class.

      Your Ans :

Solution 1 :-

a) func needs to be necessarily implemented, Entry's show is not accessible from outer Map interface. Map.max_elements would be 10 value as desired.

b) show needs to be necessarily implemented not func. Entry.max_elements would give value 5 as desired.

Code for part (a) , please complete part (b) code by yourself.

Program 5.2.c

```
import java.util.*;
interface Map
{
   int max_no = 10 ;
```

```
    interface Entry
    {
        int max_no = 5 ;
        void show();
    }
    void func();
}

class Learning implements Map
{
    int max_no = 5 ;
    public void func()
    {
        System.out.println(Map.max_no);
    }
// not necessary to implement show() in Learning

}

public class A
{
    public static void main(String[] args)
    {

        Map t = new Learning();
        t.func() ;
    }
}
```

5.3 Nested Classes

A nested class in Java is a class that is declared within another class. Nested classes are used to logically group classes in one place, making them easier to comprehend and maintain. It also has access to all of the outer class's members, including private data members and methods. It is also a member of the enclosing class or Outer class.

```
class OuterClass{
  ......
 class InnerClass{
  ......
 }
```

```
}
```

Advantages of Nested Classes:

1. Inner Class can access the members (Including private) of the outer class.
2. Logically groups the classes which increase the readability and maintainability of the code.

Before going into the types of nested class, we will see a code snippet, which gives insights into the concept of nested class.

Program 5.3.a:

```
class Outer{
   int a=20;
   void print(){
      System.out.println(a); //line1
   }
  class Inner{
     int b =19;
    void show(){
         System.out.println(b); //line2
     }
   }
}
public class Main  {
   public static void main(String[] args) {
      Outer o = new Outer();
      o.print();
      Outer.Inner i1 = new Outer.Inner(); //line3
      Outer.Inner i2 = new o.Inner(); //line4
      Outer.Inner i3 = o.new Inner(); //line5
      i1.show(); //line6
      i2.show(); //7
      i3.show(); //8
   }
}
/*
1. Just give more attention to the marked line numbers. Are there any problems with those lines? If any? What are those?

2. Out of lines 3 to 5, which is the correct declaration of the inner class object? Why is it correct and not others, state your reasons?
```

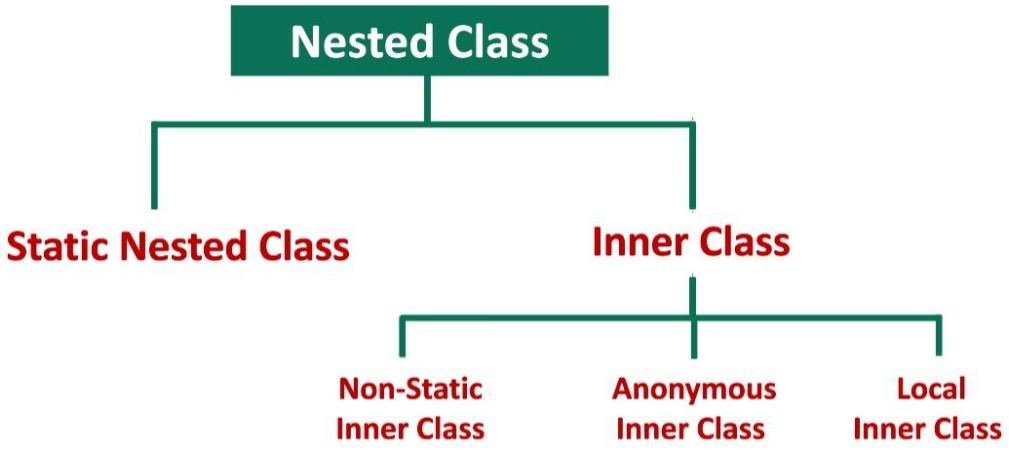
3. What are the lines that need to be commented on for the output to be:
Expected Output: 20
19
**/*

Bonus point question:
After correctly commenting on the lines as asked in Question 3,
What happens if I swap lines 1 and 2 in the above program? Will the output become as the below-expected output?
Expected output: *19*
20
Or Will there be any error? If any, why is it?

Note: *The above example refers to a type of nested class called "Member Inner Class" or "Non-Static inner class".*

5.3.1 Types of Nested Classes

1. **Static nested class**: Nested classes that are declared static are called static nested classes.
2. **Inner class**: An inner class is a non-static nested class.



5.3.2 Static Nested Classes

As we already knew from the above program 5.3.a that in order to access the instance or object of the InnerClass, we first need to create the object of the OuterClass which can be used to create the instance of InnerClass. This essentially means that the inner class object is closely associated with the OuterClass i.e, for every InnerClass object, we need to create an OuterClass object right!

Q. Can we weaken this association between OuterClass and InnerClass?
Yes, We can! Use the **static** keyword before the InnerClass body, which helps us to access the InnerClass object without even creating the OuterClass instance.

Question:
Think about the benefits of using Static Nested Classes?

Program 5.3.b:

```
class OuterClass
{
        static String name = "Teja";
        String id = "2020H1030142P";
        private static int age = 22;
        static class StaticNestedClass
        {
                void display()
                {
           //line1
                        System.out.println("Student's name is " + name );
                        System.out.println("ID is" + id); //line2
                        System.out.println("Age is" + age); //line3
                }
        }
}

public class Main
{
        public static void main(String[] args)
        {
                OuterClass.StaticNestedClass snc =
          new OuterClass.StaticNestedClass();
                snc.display();

        }
}

/*
What is the problem with the above code? State your reasons?
How to resolve the problem? So that we can expect the output as
Expected output:
Student name is Teja
```

```
ID is 2020H1030142P
Age is 22
*/
```

5.3.3 Local Inner Class

Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause.

Let us see some programs which give insights into the Local Inner classes.

```
class OuterClass{
   private String topic = "Inner Classes";
   void show(){
      String subtopic = "Local Inner Classes";
      /*____*/ class InnerClass{    //line1
      public void print(){      //line2
         System.out.println("Topic is "+ topic+"\nSub-Topic is "+subtopic);
      }
   };
   InnerClass i = new InnerClass();
   i.print();
   }
}

class Main {
   public static void main(String args[]){
      OuterClass o = new OuterClass();
      o.show();
   }
}

/*
1. Will there be any issue with giving the public access rights to the method "print()" in line2?
2. Can we prepend the "public" access modifier to the "Innerclass"? State your reasons.
*/
```

What are your observations about giving access rights to the “InnerClass” in the above program? We should think of the “InnerClass” as a piece of code inside the method. So, can we prepend access modifiers to the variables inside a method?

Program 5.3.c:

Now, we will see that a local inner class can not be local to a method, but any block like for-loop block or if-clause, we will see an example where an inner class is local to an if-clause block.

```
public class Outer
{
        public int num = 5;
        public int getNum()
        {
                return num;
        }
        public static void main(String[] args)
        {
                Outer outer = new Outer();
        System.out.print("Sum of natural numbers ");
                if(outer.getNum() > 0) //line1
                {
             /*____*/ class Inner  //line2
                        {
                                public int sumOfNumbers()
                                {
            int n = outer.getNum(); //line3
            int sum = (n*(n+1))/2;
                                        System.out.print("exists and sum is ");
                                        return sum;
                                }
        }
                        Inner inner = new Inner();
                        System.out.print(inner.sumOfNumbers());
                }
                else
                {
                        System.out.println("does not exists ");
                }
      Inner in = new Inner(); //line4
         }
}
/*
Hello chhaatron!
    1. Shall we replace "outer.getNum()" with "outer.num" in line1? What happens if we do it?
    2. Can we replace "outer.getNum()" with "num" in line3? If not, fix the code to make it work!
    3. Could we prepend the "static" type before class "Inner" in line2?
    4. What's wrong with line4?
```

```
*/
```

5.3.4 Anonymous Inner Class

These classes are inner classes without a name and for which only a single object is created. An anonymous inner class might be handy when making an instance of an object with certain “extras” such as overloading methods of a class or interface, without having to create a subclass for a class.

Let’s take an example scenario where we can plugin the concept of the anonymous inner class. Here, we have a class “CourseName” which implements the “Name” interface.

Program 5.3.d:

```
interface Name
{
        String name = "OOP";
        void getName();
}
class Main
{
        public static void main(String[] args)
        {
                CourseName obj=new CourseName();
                obj.getName();
        }
}
class CourseName implements Name
{
   public void getName()
        {
                System.out.print("Our course name is "+ name);
        }
}
```

Essentially, we have just overridden the getName method of the interface right? Then what is the need for having a sub-class “CourseName”. We can use the concept of anonymous class here.

Program 5.3.e:

```
interface Name
{
```

```
        String name = "OOP";
        void getName();
}
class Main
{
        public static void main(String[] args)
        {
    Name obj = new Name() {
      public void getName() {
         // printing course-name
        System.out.print("Our course name is "+ name);
      }
    };
                obj.getName();
        }
}
```

Now, we will have some other insights about the anonymous inner class nature. See the below program.

Program 5.3.f:

```
interface Name
{
        String name = "OOP";
        void getName();
}
class Main
{
        public static void main(String[] args)
        {
    Name obj = new Name() {
      /*____*/ final  String name = "Java"; //line1
      public void getName() {
         // printing course-name
        System.out.println("Our course name is "+ name);
        //We have to print the name variable from the interface
        //what to write here? to do so?
        /*.....*/
      }
    };
                obj.getName();
        }
}
```

```
/*
Hello!

1. What code do we need to add in the above code, to get the following output?
      Expected output:
      Our course name is Java
      Our course name is OOP
2. Can we have "static" type members inside the anonymous class?
3. Prepend line1 with "static"? Observe what happens? State your reasons?
4. Can we use the "final" keyword before "name" in line1?
*/
```

5.4 Generics

Have you ever wondered how classes like HashSet or ArrayList might have been implemented in Java? The challenge that arises while defining such classes is that the programmer doesn't know which data type would be used in advance. For example, there is no way for the programmer to know in advance whether the arraylist would be an array of integer, or double.

Solution 1: Use the Object class

Since the Object class is a super class of every other class in Java, the first solution that comes to mind would be to simply use an Object class everywhere. While the solution is correct, it raises another problem of type-checking. There is no way for the programmer to ensure that all the objects entered are indeed of the same type, as all objects are treated as instances of the Object class

Solution 2: Use of Generics

This is where Generics come into the picture. Generics is a powerful feature provided by the Java language to design and implement such classes. Let's dive into a bit more details:

5.4.1 A simple understanding

Generics mean parameterized types. The idea is to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types.

A basic skeleton to define and use generics in a class can be found here:

Program 5.4.a

```java
// A Simple Java program to show working of user defined
// Generic classes

// We use < > to specify Parameter type
// T is automatically bound with the data-type supplied during object creation
class Test<T>
{
        // An object of type T is declared
        T obj;
        Test(T obj) { this.obj = obj; } // constructor
        public T getObject() { return this.obj; }
}

// Driver class to test above
class Main
{

    // A generic function
    public static <U> void genericFunc(U element) {
        System.out.println(element.getClass().getName() + " = " + element);
    }

        public static void main (String[] args)
        {
                // instance of Integer type
                Test <Integer> iObj = new Test<Integer>(15);
                System.out.println(iObj.getObject());

                // instance of String type
                Test <String> sObj = new Test<String>("GeeksForGeeks");
                System.out.println(sObj.getObject());

        // Calling generic functions
        genericFunc(1);
        genericFunc("Hello");
         }
}

/** Points to Ponder:
Why did we do Test <Integer> iObj = new Test<Integer>(15); instead of Test <int> iObj = new Test<int>(15);
Would something like this be valid: Test <int []> iObj = new Test<int []>(15);
What does the above two experiments tell you about the nature of generics?

**/
```

5.4.2 Type Safety

Now, let's explore the most important feature provided by generics, that is type safety. Generics ensure that the code you have written is free from type mismatch errors, and such errors are detected at compile time only, instead of having to run the program to find errors.

To understand in more detail, try running the following program:

Program 5.4.b

```
// A Simple Java program to demonstrate that NOT using
// generics can cause run time exceptions
import java.util.*;

class Test
{
	public static void main(String[] args)
	{
		// Creating a an ArrayList without any type specified
		ArrayList al = new ArrayList();

		al.add("Sachin");
		al.add("Rahul");
		al.add(10); // Compiler allows this

		String s1 = (String)al.get(0);
		String s2 = (String)al.get(1);

		// Causes Runtime Exception
		String s3 = (String)al.get(2);
	}
}
```

Try to reason why the following code didn't raise any compile-time errors, what went wrong with the code during runtime, and how can it be fixed with the help of generics

A better way to write the same program would have been:

Program 5.4.c

```
// Using generics converts runtime exceptions into
// compile time exception.
import java.util.*;

class Test
{
	public static void main(String[] args)
	{
```

```
                // Creating a an ArrayList with String specified
                ArrayList <String> al = new ArrayList<String> ();

                al.add("Sachin");
                al.add("Rahul");

                // Now Compiler doesn't allow this
                al.add(10);

                String s1 = (String)al.get(0);
                String s2 = (String)al.get(1);
                String s3 = (String)al.get(2);
        }
}
```

Try running the above program, and see how it is better than the one that didn't use generics

5.4.3 Sneak Peek - Bounded Type Parameters

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.
You can explore more about bounded type parameters here.

5.4.4 Exercise

1. Try building a custom pair class that can be used to store two objects. The two objects can be of the same, or different types. Also add two methods to print each of the two objects. The class should have a user-facing interface similar to this:

Program 5.4.d

```
CustomPair<Integer, Double> p1 = new CustomPair<Integer, Double> (1, 1.0);
// Should print 1
p1.printFirst();
// Should print 1.0
p1.printSecond();
```

2. Implement a generic sort function, that sorts any given array of elements, with the only restriction that the element has implemented the comparable interface. You can read about the comparable interface here