9. Understanding Generics

IT 1406 – Introduction to Programming

Level I - Semester 1

Understanding Generics

9.1 What are Generics ?

Generics = "*Parameterized Types*"

"***Parameterized Types***" is an enrichment into Java type system that facilitate you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.

Previously, in Java generalized classes, interfaces, and methods used ***Object*** references to operate on various types of objects. **BUT** the problem was it doesn't operate with ***type safety***.

So Generics was added to **Java 5** to provide compile time type checking and to remove risk of **ClassCastException**

So what else with Generics ?

Also, Generics *detached the necessity of casting*, that translate between Object and the type of data that is actually being operated upon. Because all casts become automatic and implicit.

This streamlined process enhance the easier code *readability*, *reusability* and *faster runtime* execution.

It ensures *Compile-time Type Checking* because the compiler act as a debugger which check for the correctness of Type.

Implementation of Generics

Let’s begin our Implementation with a *Non-Generic* class

```
public class Box {
        private Object object;

        public void set(Object object) { this.object = object; }
        public Object get() { return object; }
}

public class BoxDemo1 {
        public static void main(String[] args) {  // Box for integers (?)

        Box integerBox = new Box();
        integerBox.add(new Integer(10));  // we are casting to Integer. Why?
        Integer someInteger = (Integer)integerBox.get(); //type casting, error prone and can cause ClassCastException
        System.out.println(someInteger); }
}
```

Problems Identified with *Non Generic* Box *class*

- Since, its methods accept or return an ***Object*** freely, but there is no way to verify how the classes used at compile time.

- One part of the code may enter an **Integer** into the Box and expects an **Integer** as outcome, while another part of the code may mistakenly pass a **String** in and which leads to a ***Runtime Error***.

- Notice that while using this class, we have to use type casting and it can produce ***ClassCastException*** at runtime.

Solution ?????

The Ideal Solution for the above problems are **Generics** in Java.

- It introduces formal type parameter and parameterized types to avoid classes being rebuilt for differing base types.

- Each instance of a generic class shares the same code (unlike C++ template)

- It feeds upon itself: generic type parameters can propagate.

(e.g)

List <E> within Stack <E>

A Simple Generic Class Implementation

```
// Creating a Generic Class Gen
class Gen<T> {          // T is the name of Type Parameter
                        T ob;       // declare an object of type T
            Gen(T o) {// Pass the constructor a reference to an object of type T.
                        ob = o; // both the parameter o and the member variable ob of type T are same when a Gen object is created
                        }
            T getob() {             // Return ob.
                        return ob;
                        }
void showType() {       // Show type of T.
System.out.println("Type of T is " +
ob.getClass().getName()); // getClass() returns a Class object that corresponds to the type of the class of the object on which
                        //it is called.
// getName( ) method, which returns a string representation of the class name.
            }
}
```

//Using a Generic Class GenDemo

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

                Gen<Integer> iOb; // Create a Gen reference for Integers.
                iOb = new Gen<Integer>(88); // Create a Gen<Integer> object and assign its reference to iOb.
                // Notice the use of autoboxing to encapsulate the value 88 within an Integer object.
                iOb.showType(); // Show the type of data used by iOb.
                int v = iOb.getob(); // Get the value in iOb. Notice that no cast is needed.
                System.out.println("value: " + v);
                System.out.println();

                Gen<String> strOb = new Gen<String> ("Generics Test"); // Create a Gen object for Strings.
                strOb.showType(); // Show the type of data used by strOb.
                String str = strOb.getob(); // Get the value of strOb. Again, notice // that no cast is needed.
                System.out.println("value: " + str);
        }
}
```

Output of the program

```
Type of T is java.lang.Integer
value: 88

Type of T is java.lang.String
value: Generics Test
```

** *Clearly go through the comments in the above program before going to the explanation on next slide*

Explanation on Implementation

- The above implementation defines two classes. First is the generic class **Gen**, and the second is **GenDemo**, which uses **Gen**.

- The declaration of a generic class is similar to a non-generic class declaration, the only difference is that the generic class name is followed by a type parameter section.

- The type parameter **T** indicates that it can refer to any type of class like Integer, String, Double, Character, etc. The specified type of class will store and retrieve the data of the same type.

  (E.g) If type **String** is passed to **T**, then in that instance, **ob** (an object of the type passed to **T**) will be of type **String**.

- The **GenDemo** class uses the generic **Gen** class. It first creates a version of **Gen** for integers through ***Gen<Integer> iOb;***

- Here, <**Integer>** is a *type argument* that is passed to **Gen**'s type parameter, **T**. This effectively creates a version of **Gen** in which all references to **T** are translated into references to **Integer**.

Note-

The Java compiler doesn't actually create different versions of **Gen**, or of any other generic class. Instead, the compiler removes all generic type information, substituting the necessary casts, to make your code ***behave as if*** a specific version of **Gen** were created. This process is named as **Erasure** in java.

9.2 Generics Work Only with Reference Types

Is this a correct Declaration ??

```
Gen<int> intOb = new Gen<int>(53);
```

Illegal Argument

WHY ???

WHY ??

“The type argument passed to the *Type Parameter* **MUST** be a reference type **NOT** a primitive type.”

In **Gen**, it is possible to pass any class type to **T**, but you cannot pass a primitive type to a type parameter. Therefore, the above declaration is illegal.

But its not a serious issue since you can use Type wrappers to encapsulate a primitive type where Type wrapper will be transparent through autoboxing and auto-unboxing mechanism.

9.3 Generic Types Differ Based on Their Type Arguments

*"Reference of one specific version of a generic type is **not type compatible** with another version of the same generic type."*

It's a way to improve Type Safety

Let's look at an Example.....

```
iOb = strOb; // Error
```

Though **iOb** and **strOb** are of type **Gen<T>**, they are references to different types because their type parameters differ.

9.4 How Generics Improve Type Safety

To understand this Type Safety clearly, we can implement a Non Generic Class equivalent to the Generic Program **class Gen** (Slide No-07).

Because you might think the same functionality found in the generic **Gen** class can be achieved without generics, by simply specifying **Object** as the data type and employing the proper casts

Lets begin our Implementation with two classes, First **NonGen** which is functionally equivalent to **Gen** but does not use generics. Second **NonGenDemo** which demonstrate the non-generic class

```
class NonGen {
Object ob; // ob is now of type Object

        NonGen(Object o) { // Pass the constructor a reference to an object of type Object

                ob = o;
                }

        Object getob() {
                return ob; // Return type Object.
                }

void showType() { // Show type of ob.
System.out.println("Type of ob is " +
ob.getClass().getName());
}
}
```

```
class NonGenDemo {
        public static void main(String args[]) {
                NonGen iOb;
                iOb = new NonGen(88); // Create NonGen Object and store an Integer in it. Autoboxing still occurs.
                iOb.showType(); // Show the type of data used by iOb.
                int v = (Integer) iOb.getob(); // Get the value of iOb. This time, a cast is necessary.

                System.out.println("value: " + v);
                System.out.println();

                NonGen strOb = new NonGen("Non-Generics Test"); // Create another NonGen object and store a String in it.
                strOb.showType(); // Show the type of data used by strOb.
                String str = (String) strOb.getob(); // Get the value of strOb.  Again, notice that a cast is necessary.

                System.out.println("value: " + str);
                iOb = strOb;
                v = (Integer) iOb.getob();
                }
        }
```

This compiles, but it is conceptually wrong!

Run-time error!

Downsides from Non Generic Implementation

NonGen class replaces all uses of **T** with **Object**. So it can store any type of object, like generic version.

BUT the Java compiler doesn't know the data type stored in **NonGen** which is not good for two reasons.

1. Explicit casts must be employed to retrieve the stored data.

2. Many kinds of type mismatch errors cannot be found until run time.

Let's look at these issues in depth from the implemented code……

Consider the following line of code at the end of **NonGenDemo** Class

```
iOb = strOb;
v = (Integer) iOb.getob();
```

strOb is assigned to **iOb** where **strOb** is a string object, not an integer which is **syntactically correct** because all **NonGen** references are the same, and they can refer to any other **NonGen** object.

Then why it results an Error??

Based on the second line of the code it becomes **semantically wrong.**

The return type of **getob( )** is cast to **Integer**, and then an attempt is made to assign this value to **v.**

Cont….

BUT the Java Compiler doesn’t aware that **iOb** now refers to an object that stores a **String**, not an **Integer**.

So it results a run-time exception when the cast to **Integer** is attempted.

Therefore Generics was introduced to over come this situation, which can create type-safe code in where type-mismatch errors are caught at compile time and preventing a serious bug that results in a run-time exception.

Also its possible to create “generic” code using **Object** references but its not Type-Safe and misuse of them can result run-time errors which can be converted to compile time errors through introducing Generics.

9.5 Generic Class with Two Type Parameters

A generic class can have multiple type parameters. The type parameters are separated by commas.

```
class TwoGen<T, V> { // A simple generic class with two type parameters: T and V.
        T ob1; V ob2;
        TwoGen(T o1, V o2) { // Pass the constructor a reference to an object of type T and an object of type V.
        ob1 = o1; ob2 = o2; }
        void showTypes() { // Show types of T and V.
        System.out.println("Type of T is " + ob1.getClass().getName());
        System.out.println("Type of V is " + ob2.getClass().getName());
        }
        T getob1() {
        return ob1; }
        V getob2() {
        return ob2;
        }
}
```

```
class SimpGen { // Demonstrate TwoGen.
        public static void main(String args[]) {
        TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88,
"Generics");
        tgObj.showTypes(); // Show the types.

        int v = tgObj.getob1(); // Obtain and show values.
        System.out.println("value: " + v);
        String str = tgObj.getob2();
        System.out.println("value: " + str);
                }
        }
```

Guess the Output……

Output of the program

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

Explanation on Implementation

- The above program specify two parameters T and V separated with coma defined as below,

```
class TwoGen<T, V> {
```

- Since it has two type parameters, two type arguments must be passed to **TwoGen** when an object is created it has been defined as below

```
TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88, "Generics");
```

- Here , **Integer** is substituted for **T**, and **String** is substituted for **V**.

- Likewise you can define same type arguments to **TwoGen as shown below,**

```
TwoGen<String, String> x = new TwoGen<String, String> ("A", "B");
```

- But if the type arguments are always the same, then two type parameters would be **unnecessary**

9.6 Restrictions in Generic

1. Type Parameters Can’t Be Instantiated

```
class Gen<T> {
        T ob;
        Gen() {
                ob = new T(); // Illegal!!!
        }
}
```

- The above program gives an error because in Generics you can’t create an instance of Type Parameter **T** because **T** is simply a *placeholder* and compiler doesn't know what type of object to be created.

- Therefore it is **not possible** to create an instance of a type parameter.

Cont….

2. Restrictions on Static Members

- Its Illegal to declare **static** members that use a type parameter declared by the enclosing class as shown below

```
class Wrong<T> {
        static T ob;
        static T getob() {
        return ob;
        }
}
```

Wrong !!
No static variables of type T.

Wrong !!
No static Method can use T.

Cont....

3. Generic Array Restrictions

- Restrictions applied to Generic Array are,
    1. Can’t instantiate an array whose element type is a type parameter because there is no way for the compiler to know what type of array to actually create.
    2. Can’t create an array of type-specific generic references. But this *can* create an array of references to a generic type if you use a wildcard.

4. Generic Exception Restriction

- In Generics it doesn't allow to create generic exception classes, which it cannot extend **Throwable**.