Chapter 19 :-

Introduction To Generics :

Def: is an abstract Type which is replaced by a concrete Type with out making another Class For that Type and The Generic Type provids Reusability for all concrete Types and it Can be applied as any dataType. for example we used ArrayList Generic DataStructure for make Sorting for all elements as possible as we want.

and The benefits of Generics is to enable erros To be detected at CompileTime rather Than at Runtime.

Motivations and Benefits :

a generic Type Can be in Capital letter <T> here 'T' letter is an identfier you can change it but we call That formal Generic Type and if we replaced The Generic Type by Concrete Type, we called This process Generic Instantiation

To see The Benefits of Generics we will examin an example :

ex:

[1]
```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

concrete Type

[2]
```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

→ in The first 2 lines The program will Compile fine but, it will be a RuntimeError beause string can't be Compared with a date.

→ But in the second 2 lines The program Cause a ~~Compile~~ Compile error That means That ~~The~~ error can be detected before runtime and handle it.

note: you can't replace Generic Types by primitive Types only Objects or nonprimitive Types.

Generics Provides implicit casting if it's needed.

ex:

```
ArrayList<Double> list = new ArrayList<>();
list.add(5.5); // implicit casting (autoboxing).
list.add(3.0); // implicit autoboxing!
Double doubleObject = list.get(0);
double d = list.get(1); // implicit autounboxing.
```

Defining Generic Classes and Interfaces:

now we should Know How we can write The Generics Syntax:

ex:

```
                                    declare Generic Type E
                                          ↕
public class GenericStack <E> {
                  Generic Arraylist ↔
   private java.util.Arraylist<E> list
      = new java.util.Arraylist <>();

   public int getSize() {
      return list.size(); }
                  ↘ Generic Return Type
   public E peek() {
      return list.get(list.size()-1); }
                                  ↘ Generic Parameter
   public void push(E o){
      list.add(o); }

   public E pop() {
      E o = this.peek();
Generic
object list.remove(list.getSize()-1);
      return o; }

   public boolean isEmpty(){
      return list.isEmpty(); }

   @Override
   public String toString() {
      return "Stack: " + list.toString(); }}
```

note: Don't Mislead That a Constructor Should be written in The default way without any Generics In it as following Syntax:

```
public GenericStack(){}    Correct
public GenericStack<E>(){}  incorrect
```

Occasionally, a generic class may have more Than one parameter as following Syntax:

```
<T, E, N, ---, ∞, ?>
```

note: you can define a class or an interface as a SubType of a Generic Class or interface.

Generic Methods:

A generic Type can be defined for a static method.

To declare a GenericMethod, you place The generic Type <E> immediately after the Keyword static in the method header as following Syntax:

```
public static <E> void print(E[] list){}
```

and if we want to invoke it we write it as following Syntax:

```
ClassName.<Integer> print(Integers);
```

A generic Type can be specified as a SubType of another Type and it's called **bounded** GenericType and we can write it as following Syntax :

```
<E extends ClassName>
```

ex:

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

    Rectangle rectangle = new Rectangle(2, 2);
    Circle circle = new Circle(2);
    System.out.println("Same area? " +
        equalArea(rectangle, circle));

  }

  public static <E extends GeometricObject>
  boolean equalArea(E obj1, E obj2) {

    return obj1.getArea() == obj2.getArea();

  }
```

→ Bounded GenericType

The meaning of bounded here, That E can be a finite number of Classes specially The inheritance Tree Classes of The extended Type, as the previous example GeometricObject Class.

note: The GenericType <E> is the same as <E extends Object>.

Raw Types and Backward Compatibility:

def: is a generic class or interface used without specifying a concrete Type and called RawType.
we can use it as following Syntax:

```
GenericClass gen1 = new GenericClass();
```
(RawType)

This is equivalent to the following Syntax:

```
GenericClass<Object> gen1 = new GenericClass<Object>();
```

it enables backward Compatibility (التوافق) with earlier versions of java.

you should be careful when using RawTypes because they aren't safe.

ex:

```
public class Max{
    public static void main (String[] args){
RuntimeError ← Comparable c = Max.max("Welcome", 23);
    }public static Comparable max(Comparable o1,
        Comparable o2){
      if(o1.compareTo(o2) > 0) return o1;
      else return o2;}}
```

in the preceding example, The indicated line cause a Runtime Error because we can not compare with String and integer and The program will display

-Xlint:unchecked ⇝ "Compiler Option"

we can Solve this problem by Changing The header of max method as following Syntax :

```
public static <E extends Comparable <E>> E
    max( E o1 , E o2){}
```

with that modification if there is an error it would be a Compile Error and can be checked.

Wildcard Generic Types :

def: The wild card Generic Types allows you to Change The range of Generics.

we have 3 forms of a wildcard Generic type :

? , ? extends T , ? super T

1 unbounded wildcard :- (?)

it's a wild card represents any object Type and we use it as following Syntax :

<?>

2 Bounded wild card :- (? extends T)

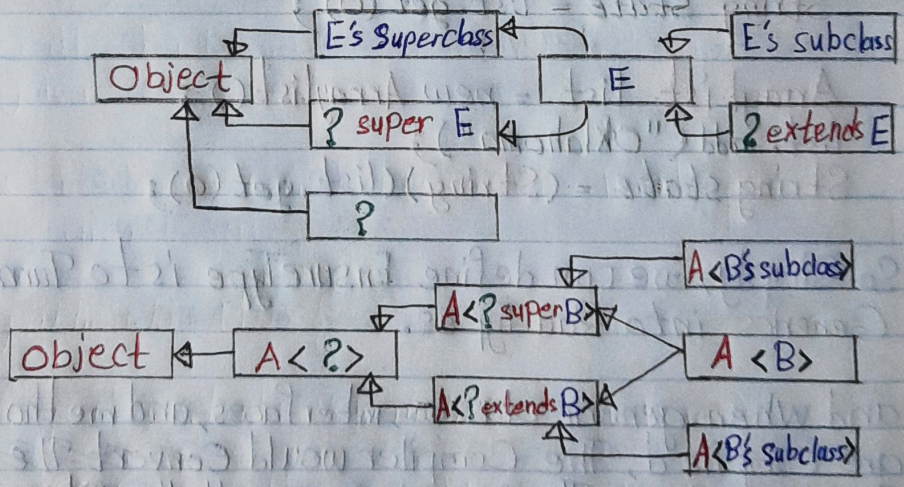
is a wildcard type that represents any object Type of (T) or subType of (T) and we can use it as following Syntax :

<? extends T>

3 Lower-bound wildCard :- (? super T)

is a wild card type that represents Type (T) or a Super Type of (T)

we will summarize all these relations in the following figure :



Ensure and Restrictions on Generics:

Def: The information on generics is used by The Compiler but is not available at Runtime This is called Type ensure.

we mean that after we use our generic information, it will be deleted and That makes the generic Code backward Compatible with the Legacy Code That uses raw Types

ex:

GenericType

Generic: Compile Time

```
ArrayList<String> list = new ArrayList<>();
list.add("Oklahoma");
String state = list.get(0);
```

RawType

RawType: RunTime

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

So Simply, we can define EnsureType is to Turn Generics into RawTypes.

and when generic classes, interfaces, and methods are Compiled, The Compiler would ~~Convert The following~~ replaces The Generic Type with the Object Type.

If a generic Type is bounded The compiler replace it by bounded Type.

note: It is important to note That a generic class is shared by all its instances regardless of its actual type (Concrete)

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<Integer> list2 = new ArrayList<>();
```

in JVM there exist one Type ArrayList So list1 instance of Arraylist and list2 instance of ArrayList.

we Kow now That The Generics are unsafe So we have To put Some Restrictions on it :

1. new E(): you can't create an instance of a generic Type. The reasone That it will be excuted at Runtime and That's denyed Generics.

2. new E[]: you can't Create an array of a generic Type. we can prevent that as following Syntax:

```
E[] elements = (E[]) new Object[Capacity];
```

3. Static<E>: A generic Type parameter of a class Is not Allowed in a static context
That all The following Syntax are illegal:

```
Public static void m(E o1) {} // Illegal
public static E o1 ; // Illegal
Static { E o2 ; } // Illegal
```

4 Exception<T>: Exception Classes Can't be Generic

That the generic can't be thrown or dosen't extend Throwable Interface.