Chapter 1

Generics

1. Introduction

- Generics - introduced by JDK5
- Designed to extend Java's type system to allow a class, interface or method to operate on objects of various types while providing compile-time type safety".
- Enables us to create classes, interfaces, and methods that will work with different kinds of data.
- Generic classes, interfaces, and methods declare type parameter/s in their definition
- e.g.

  ```
  class MyClass<T> {} //Generic class
  interface MyInterface<T> {} //Generic interface
  public <T> void MyMethod(T obj) {} //Generic method
  ```

  - T is called type parameter

1. Introduction

- Type parameters are used as a placeholder for the actual type of data that is being operated up on.
- Using generics, it is possible to create a single class, interface, or method, that automatically works with different types of data.
- Generics does not work with primitive data types.
  - ✓ Type parameters can represent only reference types—not primitive types
  - ✓ Type arguments passed to type parameter can not be primitive types.

2. Generic Class

- A generic class is a class with one or more type parameters.
- Defined as: **class** className <type-param-list> { ... }
- Objects generic classes are instantiated as:

` className <type-arg-list> varName = **new** className<> (arg-list)

- Generic classes and interfaces are collectively known as generic types.
- Each generic type defines a set of parameterized types,
- Let List is a generic class
  - ✓ List<E> is a generic type, E is type parameter.
  - ✓ List<String> is a parametrized type, String is actual type parameter or type argument.
  - ✓ List is a raw type

2. Generic Classes

- e.g. the following program defines four classes.
- Each of the first three classes define key value pair
- The fourth class uses the first three classes.
  - class Pair0 is non-generic, and the keys and values are only Integers.
    - To have key value pair of another data type - define another class
  - Pair1 is a generic class with one type parameter.
    - Key and value can be any type, but same type.
  - Pair2 is a generic class with two type parameters
    - Key and value can be any type - can be different data types.

2. Generic Classes - Example

```
class Pair0 {   //non- generic class
    Integer key, value;
    public Pair0(Integer key, Integer value){
        this.key = key;
        this.value = value;
    }
    void setKey(Integer key){ this.key = key; }
    Integer getKey(){  return key;  }
}
```

2. Generic Classes - Example

```
class Pair1<T>{  // generic class with one type parameter
    T key, value;       // member variables of type T
    public Pair1(T key, T value) {   // local variables of type T
        this.key = key;
        this.value = value;
    }
    void setKey(T key){ this.key = key;}
    T getValue(){ return value; }  //return type T
}
```

2. Generic Classes - Example

```
class Pair2<K, V> {  // A generic class with two type parameters.
    K key;                   //type parameters used to difine member variables
    V value;
    public Pair2(K key, V value){  this.key = key;     this.value = value;     }
    void setKey(K key) { this.key = key; }
    K getKey() { return key; }  //type parameters as return types
    void setvalue (V value) { this.value = value; }
    V getValue() { return value; }
}
```

2. Generic Classes - Example

```java
public class MyPairs{
    public static void main(String [] args){
        Pair0 p0 = new Pair0(9, 20);
        Pair1<Integer> pi = new Pair1<>(9, 20);
        Pair1<String> ps = new Pair1<String>("country", "Ethiopia");
        Pair2<Integer, String> pis = new Pair2<>(251, "Ethiopia");
        Pair2<String, Integer> psi = new Pair2<>("Ethiopia", 251);
        System.out.println(p2.getValue());
    }
}
```

3. Generic Interface

- Specified just like generic classes.
- e.g,

```
interface Int1 <T> { ... }
interface Int3 <T extends Number> { ... }
interface Int2 <T extends Comparable<T>> { ...
```

- A class that implements a generic interface must also be generic,

```
class Class1<T> implements Int1<T>{ ... }
class Class2<T extends Number> implements Int2<T>{ ... }
//we can use subtype of Number as uper bound
class Class3<T extends Integer> implements Int2<T>{ ... }
class Class4<T extends Comparable<T>> implements Int3<T>{ ... }
```

3. Generic Interface

- The generic class can have other type parameters.

```
class Classt5 <T, E> implements Int1<T>{ ... }
class Classy6 <T> implements Int1<T, E> { ... } // Error
```

- If a class implements a specific type of generic interface, then the implementing class does not need to be generic.

```
class Class7 implements Int1<Object> {}
class Class8 implements Int1<T> {}   //Error
class Class9 implements Int1<Integer>{}
class Class10 implements Int1<String>{}
```

4. Generic Methods

- Generic method - A method with one or more type parameter.
- Type parameters
  - ✓ precedes the method’s return type
  - ✓ can be used to declare the return type, parameters and local variables.

```
public static <T> T myMethod(T obj1) { //used to declare method `
                                        //parameter and return type
        T obj2;  //used to declare local variable
        obj2 = obj1;
        return obj2;
    }
```

4. Generic Methods

```
class Test{    //non-generic class
    static <T> T findMiddle(T [ ] arr ) { //generic metthod.
        return arr[arr.length/2];
    }
    public static void main(String [] args){
        Integer[] a1 = {1, 2, 4, 6, 7};
        String [] a2 = {"Java", "C#", "C++"};
        System.out.println(findMiddle(a1));   //type argument - Integer
        System.out.println(findMiddle(a2));   //String is type argument
     } }
```

5. Bounded Types

- Bounded types limit the parameter types that may be applied to a generic type.
- A bound is a constraint on the type of a type parameter and use the extends keyword
- In the preceding examples, the type parameters could be replaced by any class type.
- In bounded types, the type parameters could be replaced by any subclass of bound class type.
- e.g. for a class, defined as: `class C1 <T extends K> { … }`
  - ✓ The upper bound, K declares the superclass from which all type arguments must be derived.

5. Bounded Types - Example

```java
//This method returns the average of array of numbers as Double.
static <T extends Number> Double findAvg(T [] arr){
          //T subclass of abstract class Number.
       Double sum = 0.0;
       for(int i = 0; i<arr.length; i++)
          sum+= arr[i].doubleValue();
          //doubleValue is a method of Number class and
          //returns the specified value as double
       return sum/arr.length;
    }
```

5. Bounded Types - Example

```
static <T extends Comparable<T>> T findMax(T [] arr){
 //T can be any class that implements interface Comparable. i.e only objects
 //of classes that implement Comparable<T> can be used with this method.
        T max = arr[0];
        for(int i = 1; i < arr.length; i++){
            if(arr[i].compareTo(max) > 0)
                max = arr[i];
         } //compareTo is a method of Comparable that returns -1, 0. or 1
         return max;
    }
```

5. Bounded Types - Example

```
public static void main(String args[]) {
    Integer [] ai = {23, 98, 12, 4, 29};
    String [] as = {"Adama", "Addis Ababa", "Modjo", "Hawassa"};
    System.out.println(findAvg(ai));//Ok, Integer is a subclass of Number
    System.out.println(findMax(ai));//OK, Integer implements Comparable
    System.out.println(findMax(as));//Ok, String implements Comparable
    System.out.println(findAvg(as));//error.String is not subclass of Number
}
```

6. Wildcard Types

- Use wildcard when you don’t know or care what the actual type parameter is.
- Three types
  - ✓ Unbounded
  - ✓ Upper bounded
  - ✓ Lower bounded
- We can’t use type parameters with the lower bound.
- Furthermore, type parameters can have multiple bounds, while wildcards can’t.

```
<T extends Object & Comparable<T>>
```

- Bounded wildcards are used to increase API flexibility.

6. Wildcard Types - Unbounded

- Used when a method doesn't really care about the actual type.
  - ✓ any type (Object) is accepted as argument type.
- The following method prints list of any type, List<String> List<Float>

```
public void print3(List<?> list){
        for(Object o : list)
            System.out.println(o);
}
```

- The following method returns size of list of any type.

```
public int size(List<?> list){
        int count = 0;
        for(Object obj : list)   count++;
        return count;
 }
```

6. Wildcard Types - Upper-Bounded

- In generics, parameterized types are invariant.
  - e.g. List<Integer> and List<Number> are not related like class Integer and class Number are.
  - Let us define a generic method that sum up list of Number as:

```
public static double sum(List<Numer> { … }
```

  - We can not pass List<Integer> to this method as argument.
    - b/c List<Integer> is not a subtype of List<Number> or they are incompatible
  - If we redefine the method as:

```
public double sum(List<? extends Numer> { … }//upper bounded wildcard
```

  - Then we can call the method with a list of any subtype of Number class: like List<Integer>, List<Float>, ...

6. Wildcard Types - Lower-Bounded

- Specifies the lower class in the hierarchy that can be used as a generic type.
  - It is expressed using the super keyword.
- This type of bound can be used only with a wildcard
  - Type parameters don’t support lower bound.
    - <T super K> - Not supported
    - <? super K> - supported
- Let us define a generic method that adds a number to a list as:

```
static void add(List<Integer> list, Integer n){
        list.add(n);
}
```

  - Integer can be added to a list of any super type of Integer, like List<Number>
  - But we can’t call method *add* with list of any super type of Integer

6. Wildcard Types - Lower-Bounded

- ✓ If we redefine method *add* with lower bounded wildcard argument as:

```
static void add(List<? extends Integer> list, Integer n){
        list.add(n);
}
```

- ✓ We can call method *add* using the list of Integer and all of its super types (Number or Object).

- ➢ Which wildcard type to use?
  - ✓ If a parameterized type represents a T producer, use <? extends T>.
  - ✓ If it represents a T consumer, use <? super T>.
  - ✓ PECS stands for producer-extends, consumer-super.

6. Wildcard Types - Example

Consider the following class that has a member variable s of type Set<E>

```
public class MySet<E>{
        Set<E> s;
        public MySet(){
                s= new HashSet<>();
        }
}
```

1. Let us add a method to MySet class that receives list of elements and add them all to set s as follows.

```
public void addAll(List<E> src){
        for(E e : src)
                s.add(e);
}
```

6. Wildcard Types - Example

The program will not be compiled if we call method addAll as:

```
MySet<Number> msn = new MySet<>(); //msn is Set of Numbers
List<Integer> li = new ArrayList<>(); //li is list of Integers
msn.addAll(li); //error, List<Integer> & List<Number> are incompatible
```

If we rewrite method addAll using wildcard as:

```
public void addAll(List<? extends E> src){
        for(E e : src)
                s.add(e);
}
```

We can pass list of Integer types as arguments to it.

```
MySet<Number> msn = new MySet<>(); //msn is Set of Numbers
List<Integer> li = new ArrayList<>(); //li is list of Integers
msn.addAll(li); //ok, addAll expects list of subtype of Number.
```

6. Wildcard Types - Example

2. Let us add a method to MySet class that adds all elements of set s to a list.

```
public void getAll(List<E> dest){
        Iterator<E> i = s.iterator();
        while(i.hasNext())
            dest.add(i.next());
        System.out.println(dest);
    }
```

The program will not be compiled if we call this method as:

```
MySet<Integer> msi = new MySet<>(); // msi set of Integer
List<Number> ln = new ArrayList<>(); // ln List of Number
msi.getAll(ln); //error, b/c getAll() expects only list of Integer
```

6. Wildcard Types - Example

If we rewrite the method using wildcard as:

```
public void getAll(List<? super E> dest){
        Iterator<E> i = s.iterator();
        while(i.hasNext())
            dest.add(i.next());
        System.out.println(dest);
    }
```

We can call it as follows

```
MySet<Integer> msi = new MySet<>();// msi set of Integer
List<Number> ln = new ArrayList<>();// ln List of Numer
msi.getAll(ln);//Ok, b/c getAll() expects list of any super type of Integer
```

7. Generic Restrictions

- Generic types can not be instantiated.

```
class MyClass<T>{
    T o;
    public MyClass(){
        o = new T();//Illegal
    }
}
```

- No static member can use a type parameter declared by the enclosing class.

```
lass MyClass<T>{
    static T o; //error
    static T getObj(){ //error
        return obj;
    }
}
```

7. Generic Restrictions

- you cannot instantiate an array whose element type is a type parameter.

```
class MyClass<T> {
    T [] elements;
    public MyClass() {
        elements = new T[10];  //Error
    }
}
```

- A generic class cannot extend Throwable.
  - ✓ you cannot create generic exception classes.

Exercises

- Define any class with overloaded generic methods
- Write a generic method reverseArray that reverses the order of elements in an array.
- Implement a generic method that appends all elements from one array list to another. Use a wildcard for one of the type arguments.
- Implement a generic method that takes a list of any type and a target element. It returns the index of the first occurrence of the target element in the list. Return -1 if the target element cannot be found.
- Design a class that acts as a library for the following kinds of media: book, video, and newspaper.