CHP-29
Java Generics

It wouldbe nice if we could write a single sort method that could sort the elements in an Integer array, a String array or an array of any type that supports ordering.
Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.
Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods:

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.
Following are the rules to define Generic Methods:

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example:

Following example illustrates how we can print array of different type using a single Generic method:

```
public class GenericMethodTest
{
// generic method printArray
public static< E >void printArray( E[] inputArray )
{
```

CHAPTER


```
// Display array elements
for( E element : inputArray ){
System.out.printf("%s ", element );
}
System.out.println();
}
public static void main(String args[])
{
// Create arrays of Integer, Double and Character
```

```
Integer[] intArray ={1,2,3,4,5};
Double[] doubleArray ={1.1,2.2,3.3,4.4};
Character[] charArray ={'H','E','L','L','O'};
System.out.println("Array integerArray contains:");
printArray( intArray );// pass an Integer array
System.out.println("\nArray doubleArray contains:");
printArray( doubleArray );// pass a Double array
System.out.println("\nArray characterArray contains:");
printArray( charArray );// pass a Character array
}
}
```

This would produce the following result:

```
Array integerArray contains:
123456
Array doubleArray contains:
1.12.23.34.4
Array characterArray contains:
H E L L O
```

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.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example:

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects:

```
public class MaximumTest
{
// determines the largest of three Comparable objects
publicstatic<T extendsComparable<T>> T maximum(T x, T y, T z)
{
T max = x;// assume x is initially the largest
if( y.compareTo( max )>0){
max = y;// y is the largest so far
}
```



```
if( z.compareTo( max )>0){
max = z;// z is the largest now
}
return max;// returns the largest object
}
public static void main(String args[])
{
System.out.printf("Max of %d, %d and %d is %d\n\n",3,4,5, maximum(3,4,5));
System.out.printf("Maxm of %.1f,%.1f and %.1f is %.1f\n\n",6.6,8.8,7.7,
maximum(6.6,8.8,7.7));
System.out.printf("Max of %s, %s and %s is %s\n","pear",
"apple","orange", maximum("pear","apple","orange"));
}
}
```

This would produce the following result:

```
Maximum of 3,4and5is5
Maximum of 6.6,8.8and7.7is8.8
Maximum of pear, apple and orange is pear
```

Generic Classes:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters

separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example:

Following example illustrates how we can define a generic class:

```
public class Box<T>{
private T t;
publicvoid add(T t){
this.t = t;
}
public T get(){
return t;
}
public static void main(String[] args){
Box<Integer> integerBox =new Box<Integer>();
Box<String> stringBox =new Box<String>();
integerBox.add(newInteger(10));
stringBox.add(new String("Hello World"));
System.out.printf("Integer Value :%d\n\n", integerBox.get());
System.out.printf("String Value :%s\n", stringBox.get());
```



```
}
}
```

This would produce the following result:

```
IntegerValue:10
StringValue:HelloWorld
```

TUTORIALS POINT Simply Easy Learning

CHP-30
Java Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.
After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.
Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.
Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.
The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

```
public final void writeObject(Object x)throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

```
public final Object readObject()throws IOException,
ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.
To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface:

```
public class Employeeimplements java.io.Serializable
{
public String name;
public String address;
public transient int SSN;
public int number;
public void mailCheck()
{
System.out.println("Mailing a check to "+ name+" "+ address);
}
```

CHAPTER

Notice that for a class to be serialized successfully, two conditions must be met:

- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The

test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object:

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note: When serializing an object to a file, the standard convention in Java is to give the file a **.ser**extension.

```
import java.io.*;
public class SerializeDemo
{
public static void main(String[] args)
{
Employee e =new Employee();
e.name ="Reyan Ali";
e.address ="Phokka Kuan, Ambehta Peer";
e.SSN =11122333;
e.number =101;
try
{
FileOutputStream fileOut =new FileOutputStream("employee.ser");
ObjectOutputStream out=new ObjectOutputStream(fileOut);
out.writeObject(e);
out.close();
fileOut.close();
}catch(IOException i)
{
i.printStackTrace();
}
}
}
```

Deserializing an Object:

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output:

```
import java.io.*;
public class DeserializeDemo
{
public static void main(String[] args)
{
Employee e =null;
```



```
try
{
FileInputStream fileIn =new FileInputStream("employee.ser");
ObjectInputStream in=new ObjectInputStream(fileIn);
e =(Employee)in.readObject();
in.close();
fileIn.close();
}catch(IOException i)
{
i.printStackTrace();
return;
}catch(ClassNotFoundException c)
{
System.out.println("Employee class not found");
c.printStackTrace();
return;
}
System.out.println("Deserialized Employee...");
System.out.println("Name: "+ e.name);
System.out.println("Address: "+ e.address);
```

```
System.out.println("SSN: "+ e.SSN);
System.out.println("Number: "+ e.number);
}
}
```

This would produce the following result:

```
DeserializedEmployee...
Name:ReyanAli
Address:PhokkaKuan,AmbehtaPeer
SSN:0
Number:101
```

Here are following important points to be noted:

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of readObject() is cast to an Employee reference.
- The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0