Object-oriented Programming in Java

Session: 4

Generics

Objectives

- Identify the need for Generics
- List the advantages and limitations of Generics
- Explain generic class declaration and instantiation
- Define and describe generic methods
- Describe the relationship between Collection and Generics
- Explain the wildcard argument
- Describe the use of inheritance with Generics
- Describe the use of legacy code in Generics and Generics in legacy code
- Explain type inference

Introduction

- Genericity is a way by which programmers can specify the type of objects that a class can work with via parameters passed at declaration time and evaluated at compile time.
- Generic types can be compared with functions which are parameterized by type variables and can be instantiated with different type arguments depending on the context.

Generics Overview [1-3]

- Generics in Java code generates one compiled version of a generic class.
- The introduction of Generics in Java classes will help remove the explicit casting of a class object so the `ClassCastException` will not arise during compilation.
- Generics will help to remove type inconsistencies during compile time rather than at run time.
- Generics are added to the Java programming language because they enable:
  - Getting more information about a collection's type.
  - Keeping track of the type of elements a collection contains.
  - Using casts all over the program.

Generics Overview [2-3]

- Generics allow the programmer to communicate the type of a collection to the compiler so that it can be checked.
- Thus, using Generics is safe as during compilation of the program, the compiler consistently checks for the element type of the collection and inserts the correct cast on elements being taken out of the collection.

Code Snippet

```
LinkedList list = new LinkedList();
list.add(new Integer(1));
Integer num = (Integer) list.get(0);
```

In the code, an instance of linked list is created. An element of type Integer is added to the list. While retrieving the value from the list, an explicit cast of the element was required.

Generics Overview [3-3]

Code Snippet

```
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(new Integer(1));
Integer num = list.get(0);
```

In the code, the `LinkedList` is a generic class which accepts an Integer as type parameter. The compiler checks for the type correctness during compile time. It is not necessary to cast an Integer because the compiler inserts the correct cast on elements being retrieved from the list using the `get()` method.

Advantages of Generics

- Generics allow flexibility of dynamic binding.
- Generic type helps the compiler to check for type correctness of the program at the compile time.
- In Generics, the compiler detected errors are less time consuming to fix than runtime errors.
- The code reviews are simpler in Generics as the ambiguity is less between containers.
- In Generics, codes contain lesser casts and thus help to improve readability and robustness.

Limitations of Generics

- In Generics, you cannot create generic constructors.
- A local variable cannot be declared where the key and value types are different from each other.

Generic Classes [1-2]

- A generic class is a mechanism to specify the type relationship between a component type and its object type.
- The syntax for declaring a generic class is same as ordinary class except that in angle brackets (<>) the type parameters are declared.
- The declaration of the type parameters follows the class name.
- The type parameters are like variables and can have the value as a class type, interface type, or any other type variable except primitive data type.
- The class declaration such as `List<E>` denotes a class of generic type.

Generic Classes [2-2]

- The parameter to the generic class (`INTEGER in an ARRAY [INTEGER]`) is the class given in the array declaration and is bound at compile time.
- A generic class can thus generate many types, one for each type of parameter, such as `ARRAY [TREE]`, `ARRAY [STRING]`, and so on.
- Generic classes can accept one or more type parameters.
- Therefore, they are called parameterized classes or parameterized types.
- The type parameter section of a generic class can include several type parameters separated by commas.

Declare and Instantiate Generic Class [1-7]

- To create an instance of the generic class, the `new` keyword is used along with the class name except that the type parameter argument is passed between the class name and the parentheses.
- The type parameter argument is replaced with the actual type when an object is created from a class.
- A generic class is shared among all its instances.

Syntax

```
class NumberList <Element> {...}
```

Declare and Instantiate Generic Class [2-7]

Code Snippet

```
...
public class NumberList <T>
{
 private T obj;
public void add(T val)
{
  . . .
}
public static void main(String [] args)
{
 NumberList<String> listObj = new NumberList<String>
();
  . . .
}
}
...
```

Declare and Instantiate Generic Class [3-7]

- The code creates a generic type class declaration with a type variable, `T` that can be used anywhere in the class.
- To refer to this generic class, a generic type invocation is performed which replaces `T` with a value such as `String`.
- Typically, type parameter names are single, uppercase letters.
- Following are the commonly used type parameter names:
  - K - Key
  - T - Type
  - V - Value
  - N - Number
  - E - Element
  - S, U, V, and so on

Declare and Instantiate Generic Class [4-7]

The following Code Snippet illustrates how a class can be declared and initialized.

Code Snippet

```
import java.util.*;
class TestQueue <DataType> {
  private LinkedList<DataType> items = new
LinkedList<DataType>();
  public void enqueue(DataType item) {
  items.addLast(item);
}
public DataType dequeue() {
  return items.removeFirst();
}
```

Declare and Instantiate Generic Class [5-7]

```
public boolean isEmpty() {
 return (items.size() == 0);
}
public static void main(String[] args) {
 TestQueue<String> testObj = new TestQueue<>();
 testObj.enqueue("Hello");
 testObj.enqueue("Java");
 System.out.println((String) testObj.dequeue());
 }
}
```

Declare and Instantiate Generic Class [6-7]

The following Code Snippet illustrates how an instance of `TestQueue` will accept `String` as a type parameter.

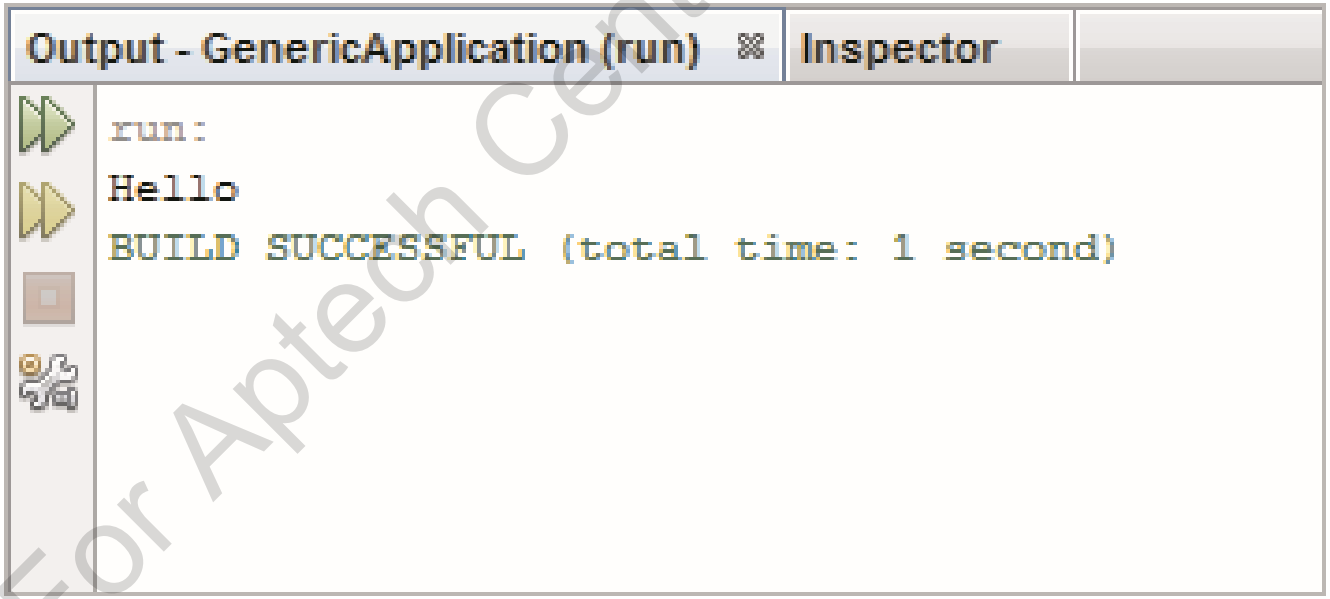
Code Snippet

```
TestQueue<String> testObj = new TestQueue<>();
```

- In the code, a generic class is created that implements the concept of queue and dequeue on any datatype such as `Integer`, `String`, and `Double`.
- The type variable or type parameter, `<DataType>`, is used for the argument type and return type of the two methods.
- Type parameters can have any name.
- Type parameters can be compared to formal parameters in subroutines.

Declare and Instantiate Generic Class [7-7]

- The name will be replaced by the actual name when the class will be used to create an instance.
- Here, `<DataType>` has been replaced by `String` within the `main()` method while instantiating the class.
- The following displays the output of the code snippet:



Generic Methods [1-7]

- Generic methods
  - Are defined for a particular method and have the same functionality as the type parameter has for generic classes.
  - Can appear in generic classes as well as in nongeneric classes.
  - Can be defined as a method with type parameters.
  - Are best suited for overloaded methods that perform identical operations for different argument type.
  - Make the overloaded methods more compact and easy to code.
- A generic method allows type parameters used to make dependencies among the type of arguments to a method and its return type.
- The return type does not depend on the type parameter, or any other argument of the method.
- This shows that the type argument is being used for polymorphism.

Generic Methods [2-7]

The following Code Snippet displays the use of a generic method.

Code Snippet

```
class NumberList <T>
{
 public<T> void display(T[] val)
 {
 for( T element : val)
 {
   System.out.printf("Values are: %s " , element);
 }
}
```

Generic Methods [3-7]

```
public static void main(String [] args)
 {
    Integer[] intValue = {1, 7, 9, 15};
    NumberList<Integer> listObj = new NumberList<> ();
    listObj.display(intValue);
 }
}
```

This code uses a generic method, `display()`, that accepts an array parameter as its argument.

Generic Methods [4-7]

The following Code Snippet demonstrates how to declare a class with two type parameters.

Code Snippet

```
import java.util.*;
public class TestQueue<DataType1, DataType2> {
 private final DataType2 num;
 private LinkedList<DataType1> items = new
LinkedList<>();
 public TestQueue(DataType2 num) {
    this.num = num;
 }
 public void enqueue(DataType1 item) {
   items.addLast(item);
 }
```

Generic Methods [5-7]

```
 public DataType1 dequeue() {
    return items.removeLast();
 }
}
```

Generic Methods [6-7]

The following Code Snippet demonstrates how to declare a nongeneric subclass.

Code Snippet

```
public class MyTest extends TestQueue<String,Integer>
{
 public MyTest(Integer num) {
    super(num);
}
 public static void main(String[] args) {
    MyTest test = new MyTest(new Integer(10));
    test.enqueue("Hello");
    test.enqueue("Java");
    System.out.println((String) test.dequeue());
 }
}
```

Generic Methods [7-7]

The following Code Snippet demonstrates the creation of a generic subclass.

Code Snippet

```
...
class MyTestQueue <DataType> extends
TestQueue<DataType> {
 public static void main(String[] args) {
   MyTestQueue<String> test = new
MyTestQueue<String>();
   test.enqueue("Hello");
   test.enqueue("Java");
   System.out.println((String) test.dequeue());
  }
}
...
```

Declare Generic Methods [1-4]

- To create generic methods and constructors, type parameters are declared within the method and constructor signature.
- The type parameter:
  - Is specified before the method return type and within angle brackets.
  - Can be used as argument types, return types, and local variable types in generic method declarations.
- There can be more than one type of type parameters, each separated by a comma.
- These type parameters act as placeholders for the actual type argument's data types, which are passed to the method.
- Primitive data types cannot be represented for type parameters.

Declare Generic Methods [2-4]

The following Code Snippet displays the generic methods present in the `Collection` interface:

Code Snippet

```
interface Collection<E>
{
public <T> boolean containsAll(Collection<T> c);
public <T extends E> boolean addAll(Collection<T> c);
}
```

- In both the methods as shown in code Snippet, `containsAll` and `addAll`, the type parameter `T` is used only once.
- For constructors, the type parameters are not declared in the constructor but in the header that declares the class.
- The actual type parameter are passed while invoking the constructor.

Declare Generic Methods [3-4]

The following Code Snippet demonstrates how to declare a generic class containing a generic constructor.

Code Snippet

```
import java.util.*;
class StudPair<T, U> {
   private T name;
   private U rollNumber;
public StudPair(T nmObj, U rollNo) {
  this.name = nmObj;
  this.rollNumber = rollNo;
}
public T displayName() {
  return name;
}
```

Declare Generic Methods [4-4]

```
public U displayNumber() {
  return rollNumber;
}
public static void main(String [] args) {
  StudPair<String, Integer> studObj = new
StudPair<>("John",2);
  System.out.println(studObj.displayName());
  System.out.println(studObj.displayNumber());
}
}
```

Accept Generic Parameters [1-3]

- A single generic method declaration can be called with arguments of different types.
- Generic methods can be defined based on the following rules:
  - Each type parameter section includes one or more type parameters separated by commas. A type parameter is an identifier that specifies a generic type name.
  - All generic method declarations have a type parameter section delimited by angle brackets preceding the method's return type.
  - A generic method's body should include type parameters that represent only reference types.
  - The type parameters can be used to declare the return type. They are placeholders for the types of the arguments passed to the generic method. These arguments are called actual type arguments.

Accept Generic Parameters [2-3]

The following Code Snippet displays a generic method declaration.

Code Snippet

```
public class GenericAcceptReturn {
 public static < E> void displayArray(E[] acceptArray)
{
  // Display array elements
   for (E element : acceptArray) {
     System.out.printf("%s ", element);
   }
     System.out.println();
 }
 public static void main(String args[]) {
  // Create arrays of Integer, Double and Character
     Integer[] intArrayObj = {100, 200, 300, 400, 500};
     Double[] doubleArrayObj = {51.1, 52.2, 53.3,
54.4};
```

Accept Generic Parameters [3-3]

```
        Character[] charArrayObj = {'J', 'A', 'V', 'A'};
        System.out.println("Integer Array contains:");
        displayArray(intArrayObj);
        System.out.println("\nDouble Array contains:");
        displayArray(doubleArrayObj);
    System.out.println("\nCharacter Array contains:");
    displayArray(charArrayObj);
    }
}
```

In the code, the `displayArray()` is a generic method declaration that accepts different type of arguments and displays them.

Return Generic Types [1-2]

- A method can also return generic data type.
- The following Code Snippet displays a method having a generic return type declaration.

Code Snippet

```
package genericreturntest;
import java.util.*;
public class GenericReturnTest {
  public static <T extends Comparable<T>> T
maxValueDisplay(T val1, T val2, T val3){
        T maxValue = val1;
        if (val2.compareTo(val1) > 0)
            maxValue = val2;
        if (val3.compareTo(maxValue) > 0)
            maxValue = val3;
  return maxValue;
}
```

Return Generic Types [2-2]

```
/**
 * @param args the command line arguments
 */
 public static void main(String[] args) {
  System.out.println(maxValueDisplay(23, 42, 1));
  System.out.println(maxValueDisplay("apples",
"oranges", "pineapple"));
 }
}
```

- In the code, the `compareTo()` method of the `Comparable` class is used to compare values which can be `int`, `char`, `String`, or any data type.
- The `compareTo()` method returns the maximum value.

Type Inference [1-5]

- Type inference enables the Java compiler to determine the type arguments that make the invocation applicable.
- It analyses each method invocation and corresponding declaration to do so.
- The inference algorithm determines the following:
  - Types of the arguments.
  - The type that the result is being returned.
  - The most specific type that works with all of the arguments.

Type Inference [2-5]

- The type arguments required to invoke the constructor of a generic class can be replaced with an empty set of type parameters (<>) as long as the compiler infers the type arguments from the context.
- The following Code Snippet illustrates this:

Code Snippet

```
Map<String, List<String>> myMap = new HashMap<String,
List<String>>();
```

Type Inference [3-5]

- In Java SE 7, the parameterized type of the constructor can be replaced with an empty set of type parameters.
- The following Code Snippet illustrates this:

Code Snippet

```
Map<String, List<String>> myMap = new HashMap<>();
```

- In the following Code Snippet, the compiler generates an unchecked conversion warning:

Code Snippet

```
Map<String, List<String>> myMap = new HashMap(); //
unchecked conversion warning
```

Type Inference [4-5]

- Java SE 7 supports limited type inference for generic instance creation.
- The type inference can be used only if the parameterized type of the constructor is apparent from the context.
- The following Code Snippet illustrates this:

Code Snippet

```
List<String> list = new ArrayList<>();
list.add("A");
// The following statement should fail since addAll
expects
// Collection<? extends String>
list.addAll(new ArrayList<>());
```

- The code does not compile.

Type Inference [5-5]

The following Code Snippet when executed compiles:

Code Snippet

```
// The following statements compile:
  List<? extends String> list2 = new ArrayList<>();
   list.addAll(list2);
```

Generic Constructors of Generic and Non-Generic Classes [1-3]

- Constructors can declare their own formal type parameters in both generic and non-generic classes.
- The following Code Snippet illustrates this:

Code Snippet

```
class MyClass<X> {
   <T> MyClass(T t) {
     // ...
   }
}
```

- The following Code Snippet shows the instantiation of the class `MyClass`.

Code Snippet

```
new MyClass<Integer>("")
```

Generic Constructors of Generic and Non-Generic Classes [2-3]

In the Code Snippet:

- The statement creates an instance of the parameterized type `MyClass<Integer>`.
- The statement specifies the type `Integer` for the formal type parameter, X, of the generic class `MyClass<X>`.
- The constructor for the generic class contains a formal type parameter, `T`.
- The compiler understands the type `String` for the formal type parameter, `T`, of the constructor of this generic class. This is because the actual parameter of the constructor is a `String` object.

Generic Constructors of Generic and Non-Generic Classes [3-3]

- The following Code Snippet is valid for Java SE 7 and later displays how compilers work:

Code Snippet

```
MyClass<Integer> myObject = new MyClass<>("");
```

- In the code, the compiler understands:
  - The type Integer is for the formal type parameter, `X`, of the generic class `MyClass<X>`.
  - The type String is for the formal type parameter, `T`, of the constructor of the generic class.

Java SE 7 Enhancements [1-2]

- Underscore characters (_) can be added anywhere between digits in a numerical literal to separate groups of digits in numeric literals.
- The `String` class can be used in the expression of a `switch` statement.
- In Java SE 7, the integral types can be defined using the binary number system.
- The Java SE 7 complier generates a warning at the declaration site of a `varargs` method or constructor with a non-reliable `varargs` formal parameter.
- The required type arguments can be replaced to invoke the constructor of a generic class with an empty set of type parameters as long as the compiler infers the type arguments from the context.

Java SE 7 Enhancements [2-2]

- A single `catch` block handles many types of exception.
- Users can define specific exception types in the `throws` clause of a method declaration because the compiler executes accurate analysis of rethrown exceptions.
- The `try-with-resources` statement declares one or more resources, which are objects that should be closed after the programs have finished working with them.
  - Object that implements the new `java.lang.AutoCloseable` interface or the `java.io.Closeable` interface can be used as a resource.
  - The statement ensures that each resource is closed at the end of the statement.

Collection and Generics [1-2]

- Collection is an object that manages a group of objects.
- Collection API depends on generics for its implementation.
- The following Code Snippet illustrates this:

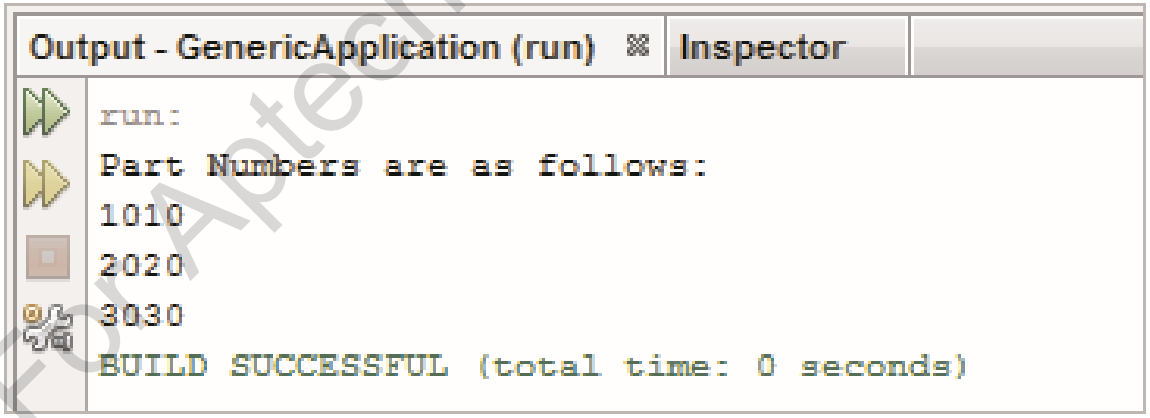
Code Snippet

```
public class GenericArrayListExample {
 public static void main(String[] args) {
   List<Integer> partObj = new ArrayList<>(3);
   partObj.add(new Integer(1010));
   partObj.add(new Integer(2020));
   partObj.add(new Integer(3030));
   System.out.println("Part Numbers are as follows:
");
   Iterator<Integer> value = partObj.iterator();
```

Collection and Generics [2-2]

```
    while (value.hasNext()) {
        Integer partNumberObj = value.next();
        int partNumber = partNumberObj.intValue();
        System.out.println("" + +partNumber);
    }
 }
}
```

The following figure displays the output of collection API and generics:



Wildcards with Generics

Exception Handling with Generics [1-2]

- Exceptions provide a reliable mechanism for identifying and responding to error conditions.
- The `catch` clause present with a `try` statement checks that the thrown exception matches the given type.
- A compiler cannot ensure that the type parameters specified in the `catch` clause matches the exception of unknown origin as an exception is thrown and caught at run time.
- Thus, the `catch` clause cannot include type variables or wildcards.
- A subclass of `Throwable` class cannot be made generic as it is not possible to catch a runtime exception with compile time parameters intact.
- In Generics, the type variable can be used in the throws clause of the method signature.

Exception Handling with Generics [2-2]

The following Code Snippet displays the use of generic type with exceptions:

Code Snippet

```
interface Command<X extends Throwable>
{
 public void calculate(Integer arg) throws X;
}
public class ExTest implements Command
<ArithmeticException>
{
 public void calculate(Integer num) throws
ArithmeticException
 {
  int no = num.valueOf(num);
  System.out.println("Value is: " + (no/0));
 }
}
```

Inheritance with Generics [1-4]

- Inheritance is a mechanism to derive new classes or interfaces from the existing ones.
- Object-oriented programming allows classes to inherit commonly used state and behavior from other classes.
- Classes can extend generic classes and provide values for type parameters or add new type parameters.
- A class cannot inherit from parametric type.
- Two instantiations of the same generic type cannot be used in inheritance.

Inheritance with Generics [2-4]

The following Code Snippet displays the use of generics with inheritance:

Code Snippet

```
class Month<T>
{
   T monthObj;
Month(T obj)
{
   monthObj = obj;
}
// Return monthObj
  T getob()
  {
     return monthObj;
  }
}
```

Inheritance with Generics [3-4]

```
// A subclass of Month that defines a second type
parameter, called V.
class MonthArray<T, V> extends Month<T>
{
  V valObj;
MonthArray(T obj, V obj2)
{
  super(obj);
  valObj = obj2;
}
V getob2()
 {
   return valObj;
 }
}
```

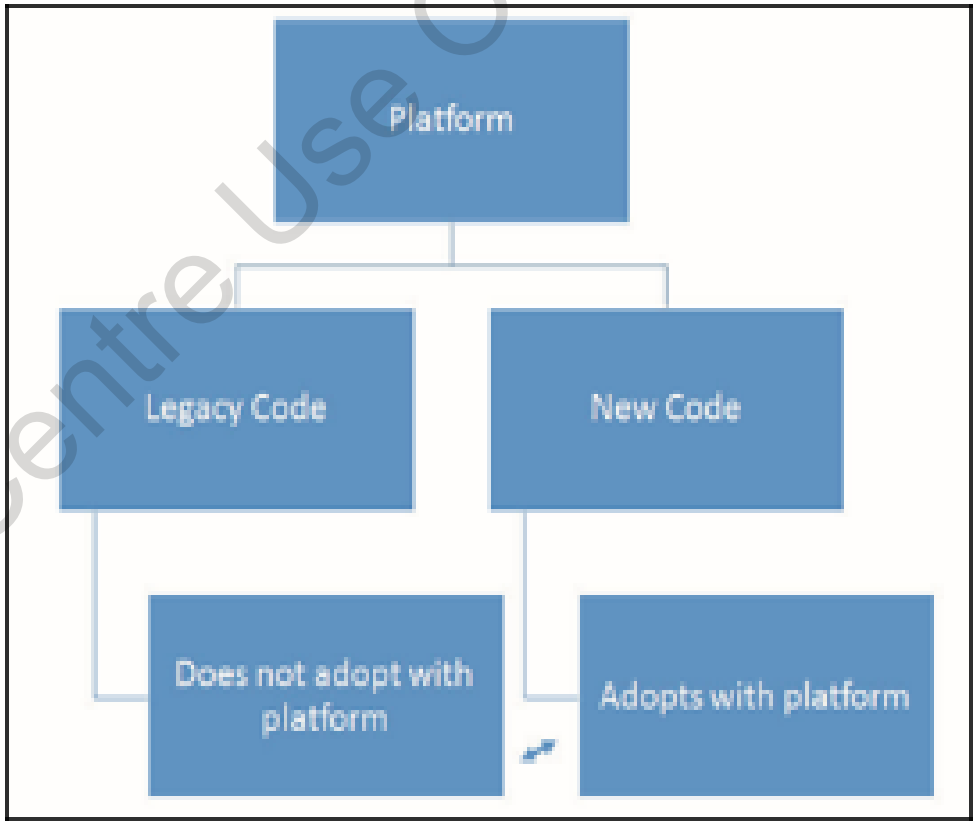
Inheritance with Generics [4-4]

```
// Create an object of type MonthArray
public class HierTest
{
  public static void main(String args[])
 {
  MonthArray<String, Integer> month;
  month = new MonthArray<>("Value is: ", 99);
  System.out.print(month.getob());
  System.out.println(month.getob2());
 }
}
```

In the code, the subclass `MonthArray` is the concrete instance of the class `Month<T>`.

Interoperability with Generics [1-4]

- In Java, genericity ensures that the same class file is generated by both legacy and generic versions with some additional information about types.
- This is known as binary compatibility as the legacy class file can be replaced by the generic class file without recompiling.



Legacy Code

Interoperability with Generics [2-4]

The following Code Snippet displays the use of legacy code with legacy client:

Code Snippet

```
import java.util.ArrayList;
import java.util.List;interface NumStack
{
    public boolean empty();
    public void push(Object elt);
    public Object retrieve();
}
```

Interoperability with Generics [3-4]

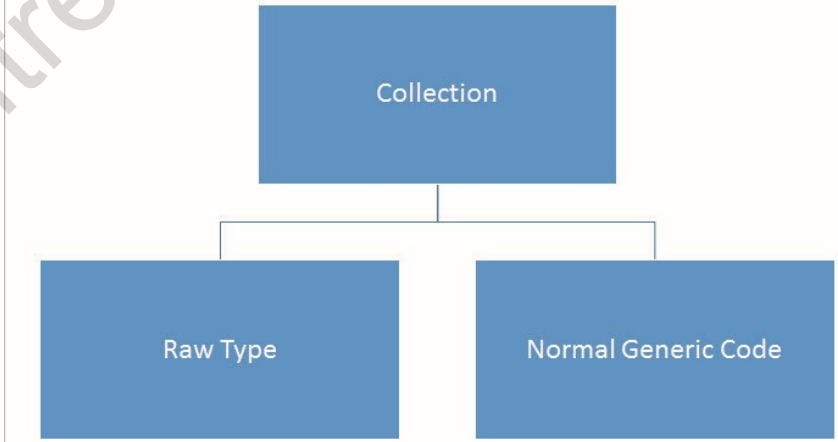
```
class NumArrayStack implements NumStack
{
   private List listObj;
public NumArrayStack()
{
   listObj = new ArrayList();}
   @Override
   public Object retrieve() {
      Object value = listObj.remove(listObj.size() -
1);
       return value;
  }
   @Override
   public String toString() {
     return "stack" + listObj.toString();
   }
}
```

Interoperability with Generics [4-4]

```
public class Client
{
  public static void main(String[] args)
  {
  NumStack stackObj = new NumArrayStack();
  for (int ctr = 0; ctr<4; ctr++)
{
   stackObj.push(new Integer(ctr));
}
  assert stackObj.toString().equals("stack[0, 1, 2,
3]");
  int top = ((Integer)stackObj.
retrieve()).intValue();
  System.out.println("Value is : " + top);
 }
}
```

Generic Library with Legacy Client [1-6]

- In generic code, the classes are accompanied by a type parameter.
- When a generic type like collection is used without a type parameter, it is called a raw type.
- A value of parameterized type can be passed to a raw type as parameterized type is a subtype of raw type.
- Java generates an unchecked conversion warning when a value of raw type is passed where a parameterized type is expected.



Generic Library with Legacy Client

Generic Library with Legacy Client [2-6]

The following Code Snippet displays the use of generic library with legacy client:

Code Snippet

```
import java.util.*;
interface NumStack
{
 public boolean empty();
 public void push(Object elt);
 public Object retrieve();
}
class NumArrayStack implements NumStack
{
 private List listObj;
 public NumArrayStack()
```

Generic Library with Legacy Client [3-6]

```
{
  listObj = new ArrayList();
}
public boolean empty()
{
  return listObj.size() == 0;
}
public void push(Object obj)
{
 listObj.add(obj);
}
public Object retrieve()
{
 Object value = listObj.remove(listObj.size()-1);
 return value; \
}
```

Generic Library with Legacy Client [4-6]

```
public String toString()
{
   return "stack"+listObj.toString();
}
}
class Client
{
  public static void main(String[] args)
 {
   NumStack stackObj = new NumArrayStack();
   for (int ctr = 0; ctr<4; ctr++)
 {
   stackObj.push(new Integer(ctr));
  }
```

Generic Library with Legacy Client [5-6]

```
  assert stackObj.toString().equals("stack[0, 1, 2,
3]");
  int top = ((Integer)stackObj.
retrieve()).intValue();
  System.out.println("Value is : " + top);
 }
}
```

When the code is compiled, an unchecked conversion warning is displayed as shown here:

```
Note - Client.java uses unchecked or unsafe operation.
Note - Recompile with -Xlint:unchecked for details.
```

Generic Library with Legacy Client [6-6]

If the code is compiled using the switch as suggested then the following message appears:

```
Client.java:21: warning: [unchecked] unchecked call to
add(E) as a member of the raw type java.util.List
listObj.add(obj);
^
1 warning
```

The warning is due to the use of the generic method add in the legacy method retrieve.

The warnings can be turned off by using the switch –`source` as shown here:

```
javac -source 1.4 Client.java
```

Erasure

- When you insert an integer into a list, and try to extract a `String` it is wrong.
- If you extract an element from list, and by casting that to String if you try to treat that as string, you will get `ClassCastException`.
- The reason is that Generics are implemented by the Java compiler as a front end conversion called erasure.
- Erasure removes all generic type information.
- All the type information between angle brackets is thrown out, so, a parameterized type like `List<String>` is converted into List.
- Type erasure maintains compatibility with Java libraries and applications which are created before generics.

Generics in Legacy Code [1-4]

- Sometimes it may be required to update the library not immediately but over a period of time.
- In such cases, the method signatures get change and consists of the type parameters.
- The method body will not change.
- This change in the method signature can be performed by making minimum changes in the method, or by creating stub or by using wrappers.
- The minimum changes that have to be incorporated are:
  - Adding type parameter to class or interface declarations
  - Adding type parameters to the class or interface which has been extended or implemented
  - Adding type parameters to the method signatures
  - Adding cast where the return type contains a type parameter

Generics in Legacy Code [2-4]

The following Code Snippet displays the use of generics:

Code Snippet

```
import java.util.*;
interface NumStack <E>
{
public boolean empty();
public void push(E elt);
public E retrieve();
}
@SuppressWarnings("unchecked")
class NumArrayStack<E> implements NumStack<E>
{
  private List listObj;
  public NumArrayStack()
  {
     listObj = new ArrayList();
  }
```

Generics in Legacy Code [3-4]

```
  public boolean empty()
  {
    return listObj.size() == 0;
  }
  public void push(E obj)
  {
      listObj.add(obj);
  }
  public E retrieve ()
  {
       Object value = listObj.remove(listObj.size()-1);
       return (E)value;
  }
  public String toString()
  {
     return "stack"+listObj.toString();
  }
}
```

Generics in Legacy Code [4-4]

Code Snippet

```
class ClientLegacy
{
  public static void main(String[] args)
  {
    NumStack stackObj = new NumArrayStack();
    for (int ctr = 0; ctr<4; ctr++)
   {
      stackObj.push(new Integer(ctr));
   }
  assert stackObj.toString().equals("stack[0, 1, 2,  3]");
  int top = ((Integer)stackObj. retrieve()).intValue();
  System.out.println("Value is : " + top);
   System.out.println("Stack contains : " +stackObj.toString());
  }
}
```

Using Generics in Legacy Code [1-4]

- A generic library should be created when there is access to source code.
- Update the entire library source as well as the client code to eliminate potential unchecked warnings.
- The following Code Snippet shows the use of generics in legacy code:

Code Snippet

```
import java.util.*;
interface NumStack <E>
{
public void push(E elt);
public E retrieve();
}
```

Using Generics in Legacy Code [2-4]

```
class NumArrayStack<E> implements NumStack<E> \
{
 private List<E> listObj;
 public NumArrayStack()
  {
      listObj = new ArrayList<E>();
  }
 public void push(E obj)
  {
      listObj.add(obj);
  }
 public E retrieve()
 {
   E value = listObj.remove(listObj.size()-1);
  return value;
 }
```

Using Generics in Legacy Code [3-4]

```
 public String toString()
 {
  return "stack"+listObj.toString();
 }
}
public class GenericClient
{
  public static void main(String[] args)
 {
   NumStack<Integer> stackObj = new NumArrayStack<Integer>();
   for (int ctr = 0; ctr<4; ctr++)
   {
     stackObj.push(ctr);
   }
   assert stackObj.toString().equals("stack[0, 1, 2, 3]");
   int top = stackObj.retrieve();
   System.out.println("Value is : " + top);
   System.out.println("Stack contains : " +stackObj.toString());
}}
```

Using Generics in Legacy Code [4-4]

- The interface and the implementing class use the type parameter.
- The type parameter <E> replaces the `Object` type from the `push()` and `retrieve()` method signature and method body.
- Appropriate type parameters are added to the client code.

Summary

- Generics in Java code generate one compiled version of a generic class.
- Generics help to remove type inconsistencies during compile time rather than at run time.
- There are three types of wildcards used with Generics such as '? extends Type', '? super Type', and '?'.
- Generic methods are defined for a particular method and have the same functionality as the type parameter have for generic classes.
- Type parameters are declared within the method and constructor signature when creating generic methods and constructors.
- A single generic method declaration can be called with arguments of different types.
- Type inference enables the Java compiler to determine the type arguments that make the invocation applicable.