Java Enterprise Edition

ANUDIP FOUNDATION

Java Collections and Generics

Objective: • Collections frameworks • Interfaces • Maps	**Materials Required:** 1. Eclipse IDE/IntelliJ/STC
Theory:40mins	**Practical:20mins**
Total Duration: 60 mins	

1Java Collections and Generics

- The Java Collections Framework
- Java Generics

1.1. The Java Collections Framework

- Java arrays have limitations.
  - They cannot dynamically shrink and grow.
  - They have limited type safety.
  - Implementing efficient, complex data structures from scratch would be difficult.
- The Java Collections Framework is a set of classes and interfaces implementing complex collection data structures.
  - A *collection* is an object that represents a group of objects.
- The Java Collections Framework provides many benefits:
  - Reduces programming effort (already there)
  - Increases performance (tested and optimized)
  - Part of the core API (available, easy to learn)
  - Promotes software reuse (standard interface)
  - Easy to design APIs based on generic collections

The Java Collections Framework consists of:

Collection interfaces
: Collection, List, Set, Map, etc.

General-purpose implementations
: ArrayList, LinkedList, HashSet, HashMap, etc.

Special-purpose implementations
: designed for performance characteristics, usage restrictions, or behavior

Concurrent implementations
: designed for use in high-concurrency contexts

Wrapper implementations
: adding synchronization, immutability, etc.

Abstract implementations
: building blocks for custom implementations

Algorithms
: static methods that perform useful functions, such as sorting or randomizing a collection

Infrastructure
: interfaces to support the collections

Array utilities
: static methods that perform useful functions on arrays, such as sorting, initializing, or converting to collections

More information on the Java Collections Framework is available at: http://download.oracle.com/javase/tutorial/collections//bookshelf/java_fundamentals_tutorial/index

1.2. The Collection Interface

- java.util.Collection is the root interface in the collections hierarchy.
    - It represents a group of Objects.
    - Primitive types (e.g., int) must be boxed (e.g., Integer) for inclusion in a collection.
    - More specific collection interfaces (e.g., List) extend this interface.
- The Collection interface includes a variety of methods:
    - Adding objects to the collection: add(E), addAll(Collection)
    - Testing size and membership: size(), isEmpty(), contains(E), containsAll(Collection)
    - Iterating over members: iterator()
    - Removing members: remove(E), removeAll(Collection), clear(), retainAll(Collection)
    - Generating array representations: toArray(), toArray(T[])

The Collection interface does not say anything about:

- the order of elements
- whether they can be duplicated
- whether they can be null
- the types of elements they can contain

This interface is typically used to pass collections around and manipulate them in the most generic way.

1.3. Iterating Over a Collection

- An *iterator* is an object that iterates over the objects in a collection.
- java.util.Iterator is an interface specifying the capabilities of an iterator.
    - Invoking the iterator() method on a collection returns an iterator object that implements Iterator and knows how to step through the objects in the underlying collection.
- The Iterator interface specifies the following methods:
    - hasNext() - Returns true if there are more elements in the collection; false otherwise
    - next() - Returns the next element
    - remove() - Removes from the collection the last element returned by the iterator

For example, to print all elements in a collection:

```
private static void print(Collection c) {
   Iterator i = c.iterator();
   while (i.hasNext()) {
      Object o = i.next();
      System.out.println(o);
   }
}
```

This can also be written as:

```
private static void print(Collection c) {
   for (Iterator i = c.iterator(); i.hasNext(); ) {
      System.out.println(i.next());
   }
}
```

And in Java 5, the Iterator can be used implicitly thanks to for-each:

```
private static void print(Collection c) {
   for (Object o : c) {
      System.out.println(o);
   }
}
```

1.4. The List Interface

- The java.util.List Interface extends the Collection interface.
- The List interface declares methods for managing an ordered collection of object (a *sequence*). You can:
    - Control where each element is inserted in the list
    - Access elements by their integer index (position in the list)
    - Search for elements in the list
    - Insert duplicate elements and null values, in most List implementations

Tip
Especially if you're dynamically resizing a collection of object, favor using a List over a Java array.

- Some additional methods provided by List include:
    - add(int index, E element) — Insert an element at a specific location (without an index argument, new elements are appended to the end)
    - get(int index) — Return an element at the specified location
    - remove(int index) — Remove an element at the specified location
    - set(int index, E element) — Replace the element at the specified location
    - subList(int fromIndex, int toIndex) — Returns as a List a *modifiable view* of the specified portion of the list (that is, changes to the view actually affect the underlying list)

1.5. Classes Implementing List

- Java provides several classes that implement the List interface.
- Two are used most commonly:

java.util.ArrayList
An ArrayList is usually your best bet for a List if the values remain fairly static once you've created the list. It's more efficient than a LinkedList for random access.
java.util.LinkedList
A LinkedList provides better performance than an ArrayList if you're frequently inserting and deleting elements, especially from the middle of the collection. But it's slower than an ArrayList for random-access.

In the following example, notice the use of the java.util.Collections.sort() static method, which does an in-place sort of the elements in a List:

```
package util;


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;


/**
 * This program reads arguments passed to it from the command-line
 * or the lines from STDIN, sorts them, and writes the result to STDOUT.
 */
public class Sort {
    public static void main(String[] args) throws IOException {
        if (args.length > 0) {
            Arrays.sort(args);
            for (int i = 0; i < args.length; i++) {
                System.out.println(args[i]);
            }
        } else {
            List lines = new ArrayList();
            BufferedReader reader = new BufferedReader(new InputStreamReader(
```

```
                System.in));
        String line = null;
        while ((line = reader.readLine()) != null) {
            lines.add(line);
        }
        Collections.sort(lines);
        for (Iterator i = lines.iterator(); i.hasNext();) {
            System.out.println(i.next());
        }
      }
   }
}
```

1.6. The Set and SortedSet Interfaces

- The java.util.Set Interface extends the Collection interface.
- The Set interface declares methods for managing a collection of objects that contain no duplicates.
    - The basic Set interface makes no guarantee of the order of elements.
    - A Set can contain at most one null element.
    - Mutable elements should not be changed while in a Set.
- The Set interface adds no methods beyond those of the Collection interface.
    - The Set interface simply enforces behavior of the collection.
- The java.util.SortedSet interface extends Set so that elements are automatically ordered.
    - A SortedSet Iterator traverses the Set in order.
    - A SortedSet also adds the methods first(), last(), headSet(), tailSet(), and subSet()

1.7. Classes Implementing Set

- Java provides several classes that implement the Set interface, including:

+ java.util.HashSet:: A hash table implementation of the Set interface. The best all-around implementation of the Set interface, providing fast lookup and updates.

java.util.LinkedHashSet
: A hash table and linked list implementation of the Set interface. It maintains the insertion order of elements for iteration, and runs nearly as fast as a HashSet.

java.util.TreeSet

A red-black tree implementation of the SortedSet interface, maintaining the collection in sorted order, but slower for lookups and updates.

```
package util;


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;


/**
 * This program computes a unique set of elements passed to it either from
 * command line (as arguments) or from STDIN (as lines).
 *
 * This demonstrates the use of Sets.
 */
public class Unique {


    public static void main(String[] args) throws IOException {
        Set unique = new HashSet(); // replace with TreeSet to get them sorted
        if (args.length > 0) {
            unique.addAll(Arrays.asList(args));
        } else {
            BufferedReader reader = new BufferedReader(new InputStreamReader(
                    System.in));
            String line = null;
            while ((line = reader.readLine()) != null) {
                unique.add(line);
            }
        }
```

```
      for (Iterator i = unique.iterator(); i.hasNext();) {
         System.out.println(i.next());
      }
   }
}
```

1.8. The Queue Interface

- The java.util.Queue interface is a collection designed for holding elements prior to processing.
    - Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.
    - Queues can implement FIFO (queue), LIFO (stack), and priority ordering.
    - Queues generally do not accept null elements.
- Queue operations include:
    - Inserting elements: add() and offer()
    - Removing and returning elements: remove() and poll()
    - Returning elements without removal: element() and peek()
- The java.util.concurrent.BlockingQueue interfaces declare additional blocking put() and take() methods, designed for concurrent access by multiple threads.

General-purpose Queue implementations:

- java.util.LinkedList
- java.util.PriorityQueue

Concurrency-specific BlockingQueue implementations:

- java.util.concurrent.ArrayBlockingQueue
- java.util.concurrent.ConcurrentLinkedQueue
- java.util.concurrent.DelayQueue
- java.util.concurrent.LinkedBlockingQueue
- java.util.concurrent.PriorityBlockingQueue
- java.util.concurrent.SynchronousQueue

1.9. The Map Interface

- The java.util.Map interface does not extend the Collection interface!
- A Map is a collection that maps key objects to value objects.
    - A Map cannot contain duplicate keys
    - Each key can map to at most one value.
- The Map interface methods include:
    - Adding key-value pairs to the collection: put(K, V), putAll(Map)

  - Retrieving a value by its key: get(K)
  - Testing size: size(), isEmpty()
  - Testing membership: containsKey(K), containsValue(V)
  - Removing members: remove(K), clear()
- The java.util.SortedMap interface extends Map so that elements are automatically ordered by key.
  - Iterating over a SortedMap iterates over the elements in key order.
  - A SortedMap also adds the methods firstKey(), lastKey(), headMap(), tailMap(), and subMap()

1.10. Retrieving Map Views

- You can also retrieve collections as *modifiable views* from the Map representing the keys (keySet()), the values (values()), and a Set of key-value pairs (entrySet()).
  - These collections are used typically to iterate over the Map elements.
  - These collections are backed by the Map, so changes to the Map are reflected in the collection views and vice-versa.
- Iterating over Map elements is done typically with a Set of objects implementing the java.util.Map.Entry interface.
  - You retrieve the Set of Map.Entry objects by invoking Map.entrySet().
  - To iterate over the Map elements using an explicit Iterator:

```
for (Iterator i = map.entrySet().iterator; i.hasNext(); ) {
    Map.Entry entry = (Map.Entry) i.next();
    System.out.println( entry.getKey() + ":\t" + entry.getValue() );
}
```

  - To iterate over the Map elements using the for-each syntax:

```
for ( Map.Entry entry: map.entrySet() ) {
    System.out.println(entry.getKey() + ":\t" + entry.getValue());
}
```

1.11. Classes Implementing Map

- Java provides several classes that implement the Map interface, including:

+ java.util.HashMap:: A hash table implementation of the Map interface. The best all-around implementation of the Map interface, providing fast lookup and updates.

java.util.LinkedHashMap

A hash table and linked list implementation of the Map interface. It maintains the insertion order of elements for iteration, and runs nearly as fast as a HashMap.

java.util.TreeMap

A red-black tree implementation of the SortedMap interface, maintaining the collection in sorted order, but slower for lookups and updates.

```
package util;


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;


/**
 * This program computes the frequency of words passed to it either as command
 * line arguments or from STDIN. The result is written back to STDOUT.
 *
 * This demonstrates the use of Maps.
 */
public class WordFrequency {


    public static void main(String[] args) throws IOException {
        // replace with TreeMap to get them sorted by name
        Map wordMap = new HashMap();
        if (args.length > 0) {
            for (int i = 0; i < args.length; i++) {
                countWord(wordMap, args[i]);
            }
        } else {
            getWordFrequency(System.in, wordMap);
```

```
    }
    for (Iterator i = wordMap.entrySet().iterator(); i.hasNext();) {
        Map.Entry entry = (Map.Entry) i.next();
        System.out.println(entry.getKey() + " :\t" + entry.getValue());
    }

}

private static void getWordFrequency(InputStream in, Map wordMap)
        throws IOException {
    BufferedReader reader = new BufferedReader(new InputStreamReader(in));
    int ch = -1;
    StringBuffer word = new StringBuffer();
    while ((ch = reader.read()) != -1) {
        if (Character.isWhitespace(ch)) {
            if (word.length() > 0) {
                countWord(wordMap, word.toString());
                word = new StringBuffer();
            }
        } else {
            word.append((char) ch);
        }
    }
    if (word.length() > 0) {
        countWord(wordMap, word.toString());
    }
}

private static void countWord(Map wordMap, String word) {
    Integer count = (Integer) wordMap.get(word);
    if (count == null) {
```

```
            count = new Integer(1);
        } else {
            count = new Integer(count.intValue() + 1);
        }
        wordMap.put(word, count);
    }
}
```

1.12. The Collections Class

- The java.util.Collections class (note the final "s") consists exclusively of static methods that operate on or return collections. Features include:
    - Taking a collection and returning an unmodifiable view of the collection
    - Taking a collection and returning a synchronized view of the collection for thread-safe use
    - Returning the minimum or maximum element from a collection
    - Sorting, reversing, rotating, and randomly shuffling List elements
- Several methods of the Collections class require objects in the collection to be comparable.
    - The object class must implement the java.lang.Comparable interface.
    - The object class must provide an implementation of the compareTo() method, which a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Note
There are several requirements for proper implementation of the compareTo() method. See the reference documentation for more information. The book *Effective Java, 2nd ed.*, by Joshua Bloch also has excellent information on the requirements and pitfalls of proper implementation of the compareTo() method.

1.13. Type Safety in Java Collections

- The Java Collections Framework was designed to handle objects of any type.
    - In Java 1.4 and earlier they used Object as the type for any object added to the collection.
    - You had to explicitly cast the objects to the desired type when you used them or else you would get compile-time errors.

```
Employee e = (Employee) list.get(0);
```

    - Worse yet, if you were dealing with a collection of objects, say of type Dog, and then accidentally added an object of an incompatible type, say a Cat, your code could

eventually try to cast the object to the incompatible type, resulting in a runtime exception.

Java arrays actually have a similar polymorphism problem that can result in runtime exceptions:

```
package com.markana.demo;

abstract class Animal {
    abstract public void speak();

    public void identify() {
        System.out.println("I'm an animal.");
    }
}

class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("woof");
    }

    @Override
    public void identify() {
        System.out.println("I'm a Dog.");
    }
}

class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("meow");
    }

    @Override
```

```
    public void identify() {
        System.out.println("I'm a Cat.");
    }
}

package com.markana.demo;

/** Demonstration that arrays are not completely type-safe
 * @author Ken Jones
 *
 */
public class ArrayTypeError {

    public static void main(String[] args) {
        // Create an array of three anonymous dogs
        Dog[] kennel = { new Dog(), new Dog(), new Dog()};

        // Let them all speak
        for (Dog d: kennel) d.speak();

        // Dogs are Objects, so this should work
        Object[] things = kennel;

        /* A Cat is an Object, so we should be able to add one to the
         * things array. Note that the following does NOT cause a
         * compiler error! Instead it throws a RUNTIME exception,
         * ArrayStoreException.
         */
        things[0] = new Cat();
    }
```

```
}
```

1.14. Java Generics

- Java Generics, introduced in Java 5, provide stronger type safety.
- Generics allow *types* to be passed as *parameters* to class, interface, and method declarations. For example:

```
List<Employee> emps = new ArrayList<Employee>();
```

- The `<Employee>` in this example is a *type parameter*.
    - With the type parameter, the *compiler* ensures that we use the collection with objects of a compatible type only.
    - Another benefit is that we won't need to cast the objects we get from the collection:

```
Employee e = emps.get(0);
```

    - Object type errors are now detected at compile time, rather than throwing casting exceptions at runtime.

1.15. Generics and Polymorphism

- What can be confusing about generics when you start to use them is that collections of a type are not polymorphic on the *type*.
    - That is, you can not assign a `List<String>` to a reference variable of type `List<Object>` (and by extension, pass a `List<String>` as an argument to a method whose parameter is type `List<Object>`); it results in a compiler error.
    - Why? If allowed, we could then add objects of an incompatible type to the collection through the more "generic" typed reference variable.
- So if you define a `printCollection()` to accept a parameter typed `List<Person>`, you can pass only `List<Person>` collections as arguments.
    - Even if Employee is a subclass of Person, a `List<Employee>` can't be assigned to a `List<Person>`.

Here's an illustration of how type parameters are not polymorphic for collections:

```
package com.markana.demo;


import java.util.*;
```

```
public class GenericsTypeError {
   public static void main( String[] args) {
      // Create a List of Dog objects
      List<Dog> kennel = new ArrayList<Dog>();

      // Adding a Dog is no problem
      kennel.add( new Dog() );

      // The following line results in a compiler error
      List<Object> objs = kennel;

      // Because if it were allowed, we could do this
      objs.add( new Cat() );

      // And now we've got a Cat in our List of Dogs
   }
}
```

1.16. Type Wildcards

- The ? type parameter *wildcard* is interpreted as "type unknown."
    - So declaring a variable as List<?> means that you can assign a List of any type of object to the reference variable.
    - However, once assigned to the variable, you can't make any assumptions about the type of the objects in the list.
- So defining your method as printCollection(List<?> persons) means that it can accept a List of any type.
    - But when invoked, you can't make any assumptions as to the type of objects that the list contains.
    - Remember, ? is "type unknown".
    - At best, you know that everything can be treated as an Object.

```
package com.markana.demo;
```

```
import java.util.*;

public class GenericsWildcardExample1 {

   public static void main( String[] args) {
      // Create a List of Dog objects
      List<Dog> kennel = new ArrayList<Dog>();

      // Adding a Dog is no problem
      kennel.add( new Dog() );

      // The following line compiles without error
      List<?> objs = kennel;

      /*
       * But now we can't make any assumptions about the type of
       * objects in the objs List. In fact, the only thing that
       * we can safely do with them is treat them as Objects.
       * For example, implicitly invoking toString() on them.
       */

      for (Object o: objs) {
         System.out.println("String representation: " + o);
      }

   }
}
```

1.17. Qualified Type Wildcards

- Declaring a variable or parameter as `List<? extends Person>`, says that the list can be of all Person objects, or all objects can be of (the same) subclass of Person.

- So you can access any existing object in the List as a Person.
- However, you can't add new objects to the List.
- The list might contain all Person objects… or Employee objects, or Customer objects, or objects of some other subclass of Person. You'd be in trouble if you could add a Customer to a list of Employees.

- Another type wildcard qualifier is super.
  - List<? super Employee> means a List of objects of type Employee or some supertype of Employee.
  - So the type is "unknown," but in this case it could be Employee, Person, or Object.
  - Because you don't know which, for "read" access the best you can do is use only Object methods.
  - But you can add new Employee objects to the list, because polymorphism allows the Employee to be treated as a Person or Object as well.
- Both extends and super can be combined with the wildcard type.

```
package com.markana.demo;

import java.util.*;

public class GenericsWildcardExample2 {

   public static void main( String[] args) {
      // Create a List of Dog objects
      List<Dog> kennel = new ArrayList<Dog>();

      // Adding a Dog is no problem
      kennel.add( new Dog() );

      /*
       * We can assign to objs a reference to any List as long
       * as it contains objects of type Animal or some subclass
       * of Animal.
       */

      List<? extends Animal> objs = kennel;

      /*
```

```
        * Now we know that the objects in the objs List are
        * all Animals or all the same subclass of Animal. So
        * we can safely access the existing objects as Animals.
        * For example, invoking identify() on them.
        */

       for (Animal o: objs) {
          o.identify();
       }

       /*
        * However, it would be a compilation error to try to
        * add new objects to the list through objs. We don't know
        * what type of objects the List contains. They might be
        * all Dogs, or all Cats, or all "generic" Animals.
        */
    }
}
```

1.18. Generic Methods

- A *generic method* is one implemented to work with a variety of types.
    - The method definition contains one or more type parameters whose values are determined when the method is invoked.
    - Type parameters are listed within `<>` after any access modifiers and before the method return type.
    - You can use any identifier for a type parameter, but single letters like `<T>` are used most often.
- For example:

```
static <T> void addToCollection(T p, Collection<T> c) {
    c.add(p);
}
```

- In this example, the object p and the objects in the collection c must all be the same type.

- You can use type wildcards, extends, and super in generic method definitions as well.

```
package com.markana.demo;

import java.util.*;

public class GenericsWildcardExample3 {

   public static <T> void add1( T obj, Collection<? super T> c) {
      c.add(obj);
   }

   public static <U, T extends U> void add2( T obj, Collection<U> c) {
      c.add(obj);
   }

   public static void main( String[] args) {

      // Create a List of Cat and Dog objects
      List<Animal> menagerie = new ArrayList<Animal>();

      // Add a Cat and a Dog
      menagerie.add( new Cat() );
      menagerie.add( new Dog() );

      // And now let's try using our generic methods to add objects
      add1( new Cat(), menagerie);
      add2( new Dog(), menagerie);

      for (Animal o: menagerie) {
         o.identify();
```

```
    }


  }
}
```

MCQ

1. What will be the output of the following Java code?

```
1.   import java.util.*;
2.   public class genericstack <E>
3.   {
4.      Stack <E> stk = new Stack <E>();
5.  public void push(E obj)
6.      {
7.         stk.push(obj);
8.  }
9.  public E pop()
10.     {
11.        E obj = stk.pop();
12.           return obj;
13.        }
14.  }
15.   class Output
16.   {
17.      public static void main(String args[])
18.      {
19.         genericstack <String> gs = new genericstack<String>();
20.         gs.push("Hello");
21.         System.out.println(gs.pop());
22.      }
23.   }
```

a)H
b)Hello
c)RuntimeError
d)CompilationError
Answer: b
Explanation: None.
Output:

```
$ javac Output.javac
```

```
$ java Output
Hello
```

2. What will be the output of the following Java code?

```
import java.util.*;
```

```
1.    public class genericstack <E>
2.    {
3.       Stack <E> stk = new Stack <E>();
4.  public void push(E obj)
5.       {
6.          stk.push(obj);
7.  }
8.  public E pop()
9.       {
10.          E obj = stk.pop();
11.             return obj;
12.          }
13.   }
14.   class Output
15.   {
16.      public static void main(String args[])
17.      {
18.         genericstack <Integer> gs = new genericstack<Integer>();
19.         gs.push(36);
20.         System.out.println(gs.pop());
21.      }
22.   }
```

a)0
b)36
c)RuntimeError
d)CompilationError
Answer: b
Explanation: None.
Output:

```
$ javac Output.javac
$ java Output
36
```

3. What will be the output of the following Java code?

```
1.    import java.util.*;
2.    public class genericstack <E>
3.    {
4.       Stack <E> stk = new Stack <E>();
```

```
5.  public void push(E obj)
6.       {
7.          stk.push(obj);
8.  }
9.  public E pop()
10.       {
11.          E obj = stk.pop();
12.             return obj;
13.          }
14.    }
15.    class Output
16.    {
17.       public static void main(String args[])
18.       {
19.          genericstack <String> gs = new genericstack<String>();
20.          gs.push("Hello");
21.          System.out.print(gs.pop() + " ");
22.          genericstack <Integer> gs = new genericstack<Integer>();
23.          gs.push(36);
24.          System.out.println(gs.pop());
25.       }
26.    }
```

a)Error
b)Hello
c)36
d)Hello36
Answer: d
Explanation: None.
Output:

```
$ javac Output.javac
$ java Output
Hello 36
```

4. What will be the output of the following Java program?

```
1.    import java.util.*;
2.    public class genericstack <E>
3.    {
4.       Stack <E> stk = new Stack <E>();
5.  public void push(E obj)
6.       {
7.          stk.push(obj);
8.  }
9.  public E pop()
10.       {
11.          E obj = stk.pop();
12.             return obj;
13.          }
```

```
14.    }
15.    class Output
16.    {
17.       public static void main(String args[])
18.       {
19.          genericstack <Integer> gs = new genericstack<Integer>();
20.          gs.push(36);
21.          System.out.println(gs.pop());
22.       }
23.    }
```

a)H
b)Hello
c)RuntimeError
d)CompilationError
Answer: d
Explanation: genericstack's object gs is defined to contain a string parameter but we are sending an integer parameter, which results in compilation error.
Output:

```
$ javac Output.javac
$ java Output
```

5. What will be the output of the following Java program?

```
1.    import java.util.*;
2.    public class genericstack <E>
3.    {
4.       Stack <E> stk = new Stack <E>();
5.  public void push(E obj)
6.       {
7.          stk.push(obj);
8.  }
9.  public E pop()
10.       {
11.          E obj = stk.pop();
12.             return obj;
13.          }
14.    }
15.    class Output
16.    {
17.       public static void main(String args[])
18.       {
19.          genericstack <Integer> gs = new genericstack<Integer>();
20.          gs.push(36);
21.          System.out.println(gs.pop());
22.       }
23.    }
```

a)H
b)Hello

c)RuntimeError
d)CompilationError
Answer: d
Explanation: generic stack object gs is defined to contain a string parameter but we are sending an integer parameter, which results in compilation error.
Output:

```
$ javac Output.javac
$ java Output
```

6. Which of these Exception handlers cannot be type parameterized?
a)catch
b)throw
c)throws
d)allofthementioned
Answer:d
Explanation: we cannot Create, Catch, or Throw Objects of Parameterized Types as generic class cannot extend the Throwable class directly or indirectly.

7. Which of the following cannot be Type parameterized?
a)OverloadedMethods
b)Genericmethods
c)Classmethods
d)Overridingmethods
Answer:a
Explanation: Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type.