LESSON IX.
Utility classes and collections

Vu Thi Huong Giang

SoICT (School of Information and Communication Technology)

HUST (Hanoi University of Science and Technology)

Objectives

- After this lesson, students can:
  - Understand the meanings of Wrapper class as well as its main purpose of conversion
  - Learn how to use mathematic constants and functions
  - Differentiate between String and StringBuffer and use appropriately
  - Understand Java Collection Framework and work with Set, List, Map data structure.
  - Understand the relations between Java Colllection and Iterator and Comparator

Content

- I. Utility classes
  - Wrapper class
  - Math class
  - String and StringBuffer
- II. Collections Framework
  - Collections Interface
  - ArrayList Class
  - HashSet Class
  - HashMap Class
- III. Iterator and Comparator Interface

I. Utility class

1. Wrapper Class

2. Math Class

3. String and StringBuffer

1. Wrapper Class-Introduction (1/2)

- Wrapper class:
  - is a wrapper around a primitive data type
  - represents primitive data types in their corresponding class instances
- Two primary purposes of wrapper classes:
  - provide a mechanism to “wrap” primitive values in an object so that the primitives can be included in activities reserved for objects
  - provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, etc.

1. Wrapper-Introduction (2/2)

- Primitive types and corresponding wrapper classes:

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

1. Wrapper class - features

- The wrapper classes are **immutable**
- The wrapper classes are **final**
- All the methods of the wrapper classes are static.
- All the wrapper classes except Boolean and Character are subclasses of an abstract class called Number
  - Boolean and Character are derived directly from the Object class

Creating Wrapper Objects with the new Operator

- All of the wrapper classes provide two constructors (except Character):
  - One takes a primitive of the type being constructed,
  - One takes a String representation of the type being constructed

```
Boolean wbool = new Boolean("false");
Boolean ybool =new Boolean(false);
Byte wbyte = new Byte("2");
Byte ybyte =new Byte(2);
Float wfloat = new Float("12.34f");
Float yfloat = new Float(12.34f);
Double wdouble = new Double("12.56d");
Double vdouble = new Double(12.56d);
Character c1 = new Character('c');
```

(Similar to Short, Integer, Long)

Creating wrapper objects by wrapping primitives using a static method

- `valueOf(String s)`
- `valueOf(String s, int radix)`
- Return the object that is wrapping what we passed in as an argument.

```
Integer i2 = Integer.valueOf("101011", 2);
// converts 101011 to 43 and assigns the value 43 to the
//Integer object i2

Float f2 = Float.valueOf("3.14f");
// assigns 3.14 to the Float object f2
```

Using Wrapper Conversion Utilities

- *primitive-type*Value(): convert the value of a wrapped numeric to a primitive
  - No-argument methods
  - 6 conversions for each six numeric wrapper class

```
Integer i2 = new Integer(42); // make a new wrapper object

byte b = i2.byteValue(); // convert i2's value to a byte primitive

short s = i2.shortValue(); // another of Integer's xxxValue methods
double d = i2.doubleValue(); // yet another of Integer's xxxValue
                             //methods
```

Converting Strings to Primitive Types

- Convert a string value to a primitive value (except Character wrapper class):
  `static <type> parse<Type>(String s)`
  - s: String representation of the value to be converted
  - <type>: primitive type to convert to (byte, short, int, long, float, double, boolean, except char)
  - <Type> : the same as <type> with the first letter uppercased

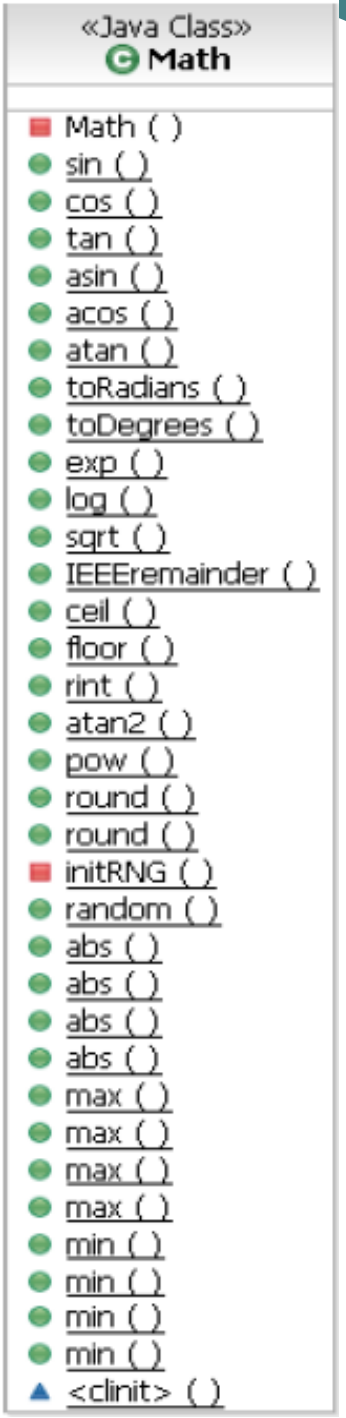
```
String s = "123";
int i = Integer.parseInt(s); //assign an int value of 123 to the int variable i
short j= Short.parShort(s) //assign an short value of 123 to the short variable j
```

I. Utility class

1. Wrapper Class
2. **Math Class**
3. String and StringBuffer

2. Math class-introduction

- The *java.lang.Math* class provides useful mathematical functions and constants.
- All of the methods and fields are static, and there is no public constructor.
- Most of methods get double parameters and return double value.
- It’s unnecessary to import any package for the *Math* class.



2. Math Class-Constants

- double E: the base of natural logarithm, 2.718...
- double PI: The ratio of the circumference of a circle to its diameter, 3.14159...

2. Math Class-Methods (1/3)

- static double abs (double x):Returns the absolute value of the argument
- static float abs (float x): Returns the absolute value of the argument
- static int abs (int x): Returns the absolute value of the argumentstatic
- static long abs (long x): Returns the absolute value of the argument

2. Math Class-Methods (2/3)

- static double acos (double): Returns the arccos of the argument (in radians), in the range (0, pi)
- static double asin (double): Returns the arcsin of the argument (in radians), in the range (-pi/2, pi/2)
- static double atan (double): Returns the arctan of the argument (in radians), in the range (-pi/2, pi/2)
- static double atan2 (double y, double x): Returns the arctan of y/x (in radians), in the range [0, 2*pi)
- static double cos (double): Returns the cosine of the argument (in radians)
- static double sin (double): Returns the sine of the argument (in radians)

2. Math Class-Methods (3/3)

- static double exp (double): Returns *e* raised to the power of the argument
- static double log (double): Returns the natural logarithm of the argumentstatic
- double pow (double base, double exp): Returns the *base* raised to the *exp* power
- static double sqrt (double): Returns the square root of the argument
- static long round (double): Returns the nearest integer to the argument
- static int round (float): Returns the nearest integer to the argument
- static double random (): Returns a pseudorandom number in the range [0, 1)

I. Utility class

1. Wrapper Class
2. Math Class
3. **String and StringBuffer**

3. String, StringBuffer & StringBuilder

- String is immutable whereas StringBuffer and StringBuilder can change their values.
- The only difference between StringBuffer and StringBuilder is that StringBuilder is unsynchronized whereas StringBuffer is synchronized. So when the application needs to be run only in a single thread then it is better to use StringBuilder. StringBuilder is more efficient than StringBuffer.
- Criteria to choose among String, StringBuffer and StringBuilder
  - If your text is not going to change use a string Class because a String object is immutable.
  - If your text can change and will only be accessed from a single thread, use a StringBuilder because StringBuilder is unsynchronized.
  - If your text can changes, and will be accessed from multiple threads, use a StringBuffer because StringBuffer is synchronous.

StringBuffer is faster than String in concatenation (1/2)

- StringBuffer is faster than String when performing simple concatenations
  - Ex1:

```
String str = new String ("First ");
str += "Second";
```

  - Ex2:

```
StringBuffer str = new StringBuffer ("First ");
str.append("Second");
```

StringBuffer is faster than String in concatenation (2/2)

- Steps in Ex 1: (Create 3 objects)
  - Create a String object
  - Create a temporary StringBuffer object
  - Append the StringBuffer object to String value
  - Convert StringBuffer object to String object

```
String str = new String ("First ");
str += "Second";
```

- Steps in Ex 2: (Create 1 object)
  - Create a StringBuffer object
  - Append the StringBuffer object to String value
- → Ex 2 is faster than Ex 1

```
StringBuffer str = new StringBuffer ("First ");
str.append("Second");
```

StringBuffer operations

- append(boolean b): StringBuffer
- append(float f): StringBuffer
- append(String str): StringBuffer
- append(StringBuffer sb): StringBuffer
- etc.
- charAt(int index): StringBuffer
- delete(int start, int end): StringBuffer
- insert(int offset, float f): StringBuffer
- insert(int offset, String str): StringBuffer
- length(): int
- etc.

II. Collections framework

1. Introduction

2. Interfaces

3. Implementations

1. Introduction

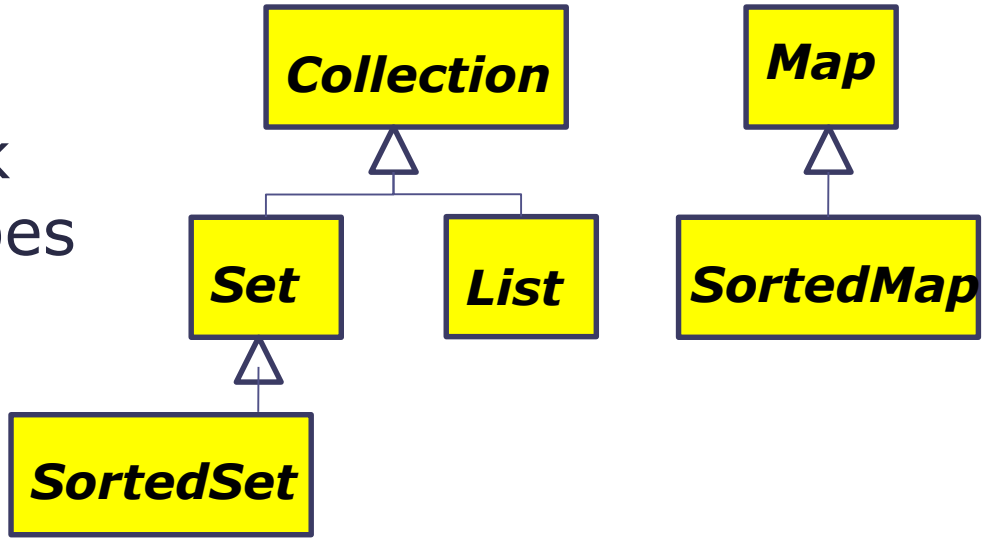
- Collection: groups multiple elements into a single unit.
- Java Collections Framework:
  - represents and manipulates collections.
  - provides:
    - A standard common programming interface to many of the most common abstraction.
    - A system for organizing and handling collections.
  - is based on:
    - Interfaces: common collection types
    - Classes: implementations of the Interfaces
    - Algorithms: data and behaviors need when using collections i.e. search, sort, iterate etc.

Advantages of the Collections Framework

- Easy to learn, easy to use the new APIs
- Foster software reuse
- Reduce the efforts to design new APIs
- Reduce the programming efforts
- Increase performance
- Provide interoperability between the unrelated APIs
- Provide resizable capability

Interfaces hierarchy

- Each interfaces in Collections Framework describes different types of functionalities



Interface name	Ordered	Allows duplicates	Maps key to object
Collection	No	Yes	No
Set	No	No	No
List	Yes	Yes	No
Map	No	No	Yes
SortedSet	Yes	No	No
SortedMap	Yes	No	Yes

II. Collection framework

1. Introduction
2. **Interfaces**
3. Implementations

a. Collection Interface

- **Collection**:
  - root interface for Java collections hierarchy.
  - extended by **List**, **Set** & **SortedSet** interfaces
  - used to represent a group of objects, or elements
  - behaviours:
    - Basic Operations
    - Bulk Operations
    - Array Operations

a. Collection Interface-Methods

```
public interface Collection {
        // Basic Operations
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(Object element);
        boolean remove(Object element);
        Iterator iterator();

        // Bulk Operations
        boolean addAll(Collection c);
        boolean removeAll(Collection c);
        boolean retainAll(Collection c);
        ….
        // Array Operations
        Object[] toArray();
        Object[] toArray(Object a[]);
    }
```

b. Set interface

- A Set is a collection that cannot contain duplicate elements
- Examples:
  - Set of cars:
    - {BMW, Ford, Jeep, Chevrolet, Nissan, Toyota, VW}
  - Nationalities in the class
    - {Chinese, American, Canadian, Indian}

b. Set Interface-Methods

- Same as Collection Methods but the contract is different:
  - No duplicates are maintained

c. SortedSet interface

- SortedSet interface:
  - extends **Set** interface.
  - maintains its elements in ascending order.
  - contains no duplicate elements
  - permits a single element to be **null.**
- In addition to methods of the Set interface, it also provides two following methods:
  - **first( ):** returns the first (lowest) element currently in the collection
  - **last( ) :** returns the last (highest) element currently in the collection

d. List Interface

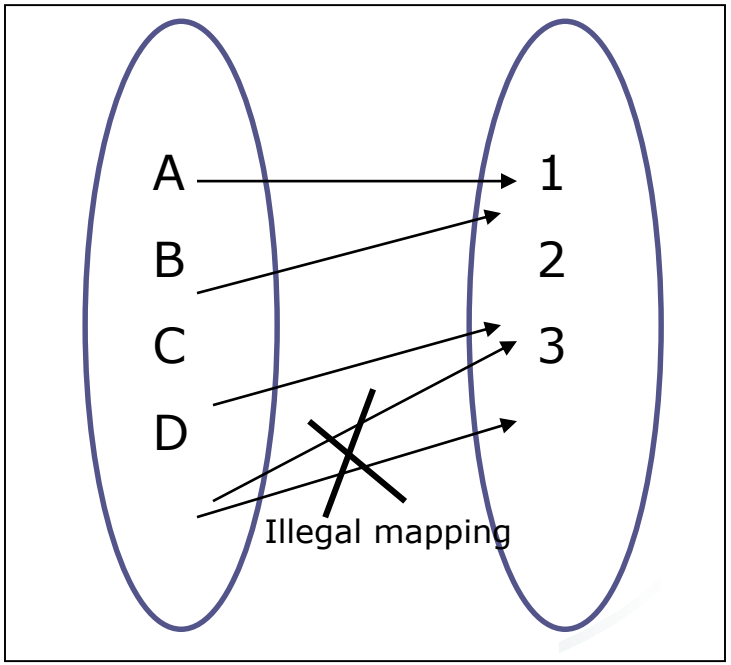
- A List is an ordered collection (sometimes called a sequence)
- Lists can contain duplicate elements
- Examples:
  - List of first name in the class sorted by alphabetical order:
    - Eric, Fred, Fred, Greg, John, John, John
  - List of cars sorted by origin:
    - Ford, Chevrolet, Jeep, Nissan, Toyota, BMW, VW

d. List Interface-Methods

- Inherits from Collection
- Some additions:
  - void add(int index, Object element);
  - boolean addAll(int index, Collection c);
  - Object get(int index);
  - Object remove(int index);
  - Object set(int index, Object element);
  - int lastIndexOf(Object o);
  - int indexOf(Object o);

e. Map interface

- A Map is an object that maps keys to values. Maps cannot contain duplicate keys.
- Each key can map to at most one value
- Examples:
  - Think of a dictionary:
    - word <-> description
  - address book
    - name <-> phone number



Map

e. Map Interface-Methods

- Basics
  - Object put(Object key, Object value);
  - Object get(Object key);
  - Object remove(Object key)
  - int size();
  - ...
- Bulk
  - void putAll(Map t);
  - void clear();
- Collection Views
  - public Set keySet();
  - public Collection values();
  - public Set entrySet();

f. SortedMap Interface

- **SortedMap** interface:
  - extends the **Map** interface
  - maintains its elements in ascending order.
  - similar to a **SortedSet** except, the sort is done on the map keys.
- In addition to methods of the Map interface, it provides two methods shown as:
  - **firstKey( )**: returns the first (lowest) value currently in the map
  - **lastKey( )**: returns the last (highest) value currently in the map

II. Collections framework

1. Introduction
2. Interfaces
3. **Implementations**

Implementations

Inter faces	Implementations				
	Hash Table	*Resizable Array*	*Balanced Tree*	*Linked List*	*Hash table + Linked list*
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

a. Set Implementations

- HashSet:
  - stores elements in a hash table
  - doesn't allow to store duplicate collections
  - permits the **null** element
  - is the best way to perform **set** implementation
- TreeSet:
  - used to extract elements from a collection in a sorted manner.
  - elements added to a TreeSet are sortable in an order.
  - It is generally faster to add elements to a **HashSet**, then converting the collection to a **TreeSet** for sorted traversal
- LinkedHashSet:
  - is the implementation of both the Hash table and Linked list
  - extends HashSet and implements **Set** interface.
  - it differs from **HashSet** that it maintains a doubly-linked list.
  - the orders of its elements are based on the order in which they were inserted into the set (insertion-order)

a. Set Implementation-Example

```
import java.util.*;
public class SetDemo {
  public static void main(String args[]) {
     int count[]={34, 22,10,60,30,22};
     Set<Integer> set = new HashSet<Integer>();
     try{
        for(int i=0; i<5; i++){
           set.add(count[i]);
        }
        System.out.println(set);

        TreeSet sortedSet=new Tree
        System.out.println("The sc
        System.out.println(sortedSet);

        System.out.println("The First element of the set is: "+
                                        (Integer)sortedSet.first());
        System.out.println("The last element of the set is: "+
                                        (Integer)sortedSet.last());
     }
     catch(Exception e){}
  }
}
```

RESULTS:

[34, 22, 10, 30, 60]
The sorted list is:
[10, 22, 30, 34, 60]
The First element of the set is: 10
The last element of the set is: 60

b. List Implementations

- **ArrayList, Vector**: resizable-array implementation of **List**.
  - Objects of Vector are synchronized by default
  - Vector is from Java 1.0, before collections framework
  - ArrayList is better and preferable than Vector
- LinkListed: a linked-list implementation of interface List
  - can be used to create stacks, queues, trees, deques

b. List Implementations-Example

```
import java.util.*;
public class ListQueueDemo {
  public static void main(String args[]) {
    int numbers[]={34, 22,10,6
    List <Integer>list = new A
    try{
      for(int i=0; i<5; i++) l
      System.out.println("the
      System.out.println(list)
      LinkedList<Integer> queu
      for(int i=0; i<5; i++) queue.addFirst(numbers[i]);
      System.out.println("The Oueue is: ");
      System.out.println(queue);
      queue.removeLast();
      System.out.println("Last element after removing"+ queue);
    }
    catch(Exception e){}
  }
}
```

RESULTS:

the List is:
[34, 22, 10, 60, 30]
The Oueue is:
[30, 60, 10, 22, 34]
After removing last element the queue is: [30, 60, 10, 22]

c. Map implementations

- **HashMap:**
  - used to perform some basic operations such as **inserting**, **deleting**, and **locating** elements in a Map
- **TreeMap:**
  - is useful when we need to traverse the keys from a collection in a sorted manner.
  - the elements added to a TreeMap must be sortable in order to work properly.
  - it may be faster to add elements to a **HashMap** , then convert the map to a **TreeMap** for traversing the sorted keys.
- **LinkedHashMap:**
  - extends HashMap with a linked list implementation that supports ordering the entries
  - entries in a LinkedHashMap can be retrieved in:
    - Insertion order - the order in which they were inserted
    - Access order - the order in which they were last accessed, from least recently accessed to most recently

c. Map implementations-Example

```
public static void main(String args[]) {
   String days[]={"Sunday", "Monday", "Tuesday", "Wednesday"
   Map map = 
   for(int i=
      map.put
   TreeMap tM
   //Rerievin
   System.out
   //Rerievin
   System.out
   //Rerievin
   System.out
   //Removing
   System.out
   System.out
   System.out
   //Rerievin
   System.out
   //Removing
   System.out
   System.out
   System.out.println("Now the tree map contain: " + tMap.values());
  }
 }
```

RESULTS:

```
Keys of tree map: [0, 1, 2, 3, 4, 5, 6]
Values of tree map: [Sunday, Monday, Tuesday,
Wednesnday, Thursday, Friday, Saturday]
First key: 0 Value: Sunday

Removing first data: Sunday
Now the tree map Keys: [1, 2, 3, 4, 5, 6]
Now the tree map contain: [Monday, Tuesday,
Wednesnday, Thursday, Friday, Saturday]

Last key: 6 Value: Saturday

Removing last data: Saturday
Now the tree map Keys: [1, 2, 3, 4, 5]
Now the tree map contain: [Monday, Tuesday,
Wednesnday, Thursday, Friday]
```

III. Iterator and comparator interface

1. Iterator Interface
2. Comparator Interface

1. Iterator Interface

- Iterator: used to move through a sequence of objects and select each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence
- operators with Iterator:
  - iterator( ): ask a container to hand an Iterator
  - next( ): get the next object in the sequence
  - hasNext( ): see if there are any more objects in the sequence
  - remove( ): remove the last element returned by the iterator

1. Iterator Interface

The interface definition:

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Sample code:

```
Collection c;
// Some code to build the collection
Iterator i = c.iterator();
while (i.hasNext()) {
  Object o = i.next();
  // Process this object
}
```

2. java.util.Comparator

- The java.util.Comparator interface can be used to create objects to pass to sort methods or sorting data structures.
- A Comparator must define a compare function which takes two Objects and returns a -1, 0, or 1
- Comparators not needed if there's a natural sorting order

Example-Person class

```
class Person{
    private int age;
    private String name;

    public void setAge(int age){
        this.age=age;
    }
    public int getAge(){
        return this.age;
    }
    public void setName(String name){
        this.name=name;
    }
    public String getName(){
        return this.name;
    }
}
```

Example-AgeComparator class

```
class AgeComparator implements Comparator{

    public int compare(Object ob1, Object ob2){
        int ob1Age = ((Person)ob1).getAge();
        int ob2Age = ((Person)ob2).getAge();

        if(ob1Age > ob2Age)
            return 1;
        else if(ob1Age < ob2Age)
            return -1;
        else
            return 0;
    }
}
```

Example-ComparatorExample class (1/2)

```
public class ComparatorExample {
   public static void main(String args[]) {
      ArrayList<Person> lst = new ArrayList<Person>();
      Person p = new Person();
      p.setAge(35);
      p.setName("A");
      lst.add(p);

      p = new Person();
      p.setAge(30);
      p.setName("B");
      lst.add(p);

      p = new Person();
      p.setAge(32);
      p.setName("C");
      lst.add(p);
```

Example-ComparatorExample class (1/2)

```
        System.out.println("Order of person before sorting is");
        for (Person person : lst) {
            System.out.println(person.getName() +
                        "\t" + person.getAge());
        }

        Collections.sort(lst, new AgeComparator());
        System.out.println("\n\nOrder of person" +
                        "after sorting by person age is");

        for (Iterator<Person> i = lst.iterator(); i.hasNext();) {
            Person person = i.next();
            System.out.println(person.getName() + "\t" +
                        person.getAge());
        }//End of for
    }//End of main
}//End of class
```

Quick quiz (1)

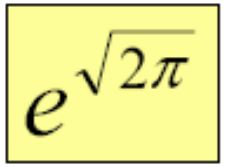
1. Which classes are immutable?
   a. String
   b. StringBuffer
   c. StringBuilder
   d. Wrapper class
2. Which classes have more than 1 constructor
   a. Integer
   b. Long
   c. Character
   d. String
   e. Boolean
3. Can you list several methods of the wrapper classes which are not static

Quick quiz (2)

4. Which command is incorrect
    a. short j = Integer.parseShort(“5”);
    b. Int j = Integer.parseInt(“15”);
    c. int k = Integer.parseInteger(“20”);
    d. double d = Double.parseDouble(“20.7f”)
    e. int h = Long.parseLong(“12”);
    f. int n = Short.parseShot(“24”);
5. Which interface allow duplicates:
    a. Set
    b. List
    c. Collection
    d. Map
    e. SortedSet
    f. SortedMap
6. In Map interface, can a key map to more than one value

Quiz 1

- Calculate the following value by two different functions provided from java.lang.Math:



Quiz 1-Solution

```
public class Calculate {

   public static void main(String[] args) {

     System.out.println(Math.pow(Math.E,
                  Math.sqrt(2.0 * Math.PI)));

     System.out.println(Math.exp(Math.sqrt(2.0 *
                  Math.PI)));
   }
}
```

Quiz 2

- Write a utility method: to remove all non-digit characters from a String
  - public static String removeNonDigits(String text)

Quiz 2-Solution

```
public class Utility {
    public static String removeNonDigits(String text) {
        int length = text.length();
        StringBuffer buffer = new StringBuffer(length);
        for (int i = 0; i < length; i++) {
          char ch = text.charAt(i);
          if (Character.isDigit(ch)) buffer.append(ch);
        }
        return buffer.toString();
    }
    public static void main(String[] args) {
        String testText = "Hello 1To2The3 Wor3ld from Guate4mala5";
        System.out.println("Response: " +
                                Utility.removeNonDigits(testText));
    }
}
```

Quiz 3

- Build an ArrayList of Float (represent an angle measured in degrees) and then sort it descendingly according to it ***sin*** value

Quiz 3-Solution-SinComparator

```
import java.util.*;
class SinComparator implements Comparator {
    public int compare(Object ob1, Object ob2) {
        float f1 = ((Float) ob1).floatValue();
        float f2 = ((Float) ob2).floatValue();

        double d1 = Math.sin(f1 * Math.PI / 180);
        double d2 = Math.sin(f2 * Math.PI / 180);

        if (d1 < d2)
          return 1;
        else if (d1 > d2)
          return -1;
        else
          return 0;
    }
}
```

Quiz 3-Solution-ListDemo

```
import java.util.*;
public class ListDemo {
    public static void main(String args[]){
        ArrayList<Float> lst = new ArrayList<Float>();
        lst.add(new Float(30));lst.add(new Float(60));
        lst.add(new Float(90));lst.add(new Float(120));
        lst.add(new Float(150));lst.add(new Float(180));

        System.out.println("Order before sorting:");
        for (Float f: lst){
          System.out.println(f);
        }
        Collections.sort(lst, new SinComparator());
        System.out.println("Order after sorting:");
        for (Iterator<Float> i = lst.iterator(); i.hasNext();) {
          Float f = i.next(); System.out.println(f);
        }
    }
}
```

Review

- Utility classes
  - Wrapper class:
    - represents primitive data types in their corresponding class instances,
    - provides utility functions for primitives
  - java.lang.Math class :
    - provides useful mathematical functions and constants
  - String and StringBuffer
    - String is immutable.
    - StringBuffer is mutable.
    - StringBuffer is faster than String in concatenation

Review

- Collections Framework
  - Interfaces: Map, Set and List.
  - Classes: ArrayList, HashSet, HashMap
- Iterator and Comparator Interface
  - Iterator: traverses all elements of a Collection.
  - Comparator: compares two different objects.