Classes – Part II

- Generic classes

- Inner classes
  - Ordinary inner class
  - Local inner class
  - Anonymous inner class
  - Event handling with Lambda expression

Generic Programming

- Allows us to write code that can be reused for objects of many different types.
- Old style of Generic programming: Using polymorphism
  - For example, the same class ArrayList can be reused for storing String and File objects.
  - Assume that ArrayList.add(Object o)

```
// Usage #1: storing String objects
ArrayList stringList = new ArrayList() ;

stringList.add(“str1”) ; // String is a descent of Object
stringList.add(“str2”) ;
```

```
// Usage #2: storing File objects
ArrayList fileList = new ArrayList() ;

fileList.add(new File(“…”)) ; // String is a descent of Object
fileList.add(new File(“…”)) ;
```

Generic Programming Using Polymorphism before Java 5(Oct. 2004)

- We can write a code that can allow different types with **Object** and **casts**

```
public class ArrayList {
  public Object get(int i) {…}
  public void add(Object o) {…}
  …
  private Object[] elementData ;
}
```

```
ArrayList filenames = new ArrayList() ;
filenames.add(new String(“a.txt”)) ;
String filename = (String) filenames.get(0) ;

filenames.add(new File(“…”)) ;
```

- What are the problems with the code?
  1. Object type **should be casted into** the proper type; Object ➔ String
  2. Some problematic codes CANNOT be checked by the compiler

     It may be a problem for the ArrayList to hold String and File at the same time!

- What can be a solution to these problems ?

  ➔ Generic Programming by Generic class (Template class)

Generic Programming Using Generic Class Since Java 5

- Generics are similar to template in C++.
- They make your programs easier to read and safer.

```
// Since Java 5
public class ArrayList <T> {
  public T get(int i) {…}
  public void add(T o) {…}
  …
  private T[] elementData ;
}
```

Type parameter

```
ArrayList<String> filenames = new ArrayList<String>() ;
filenames.add(new String(“a.txt”)) ;
String filename = filenames.get(0) ; // casting is not necessary !

filenames.add(new File(“…”)) ; // compile-time error is issued !
// The method add(String) in the type ArrayList<String> is not applicable
// for the arguments (File)
```

Generic Class: Another Example

```
class Pair<T> {
    public Pair() { first = null; second = null; } // Actually, this body is not necessary !
    public Pair(T first, T second) { this.first = first;  this.second = second; }
    public T getFirst() { return first; }
    public T getSecond() { return second; }
    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }

    private T first, second;
}
public class PairTest1 {
    public static void main(String[] args) {
        Pair<String> strPair = new Pair<String>() ;
        strPair.setFirst("Name") ;
        strPair.setSecond("Value");
        System.out.println( strPair.getFirst() + " " + strPair.getSecond()) ;

        Pair<Rectangle> recPair = new Pair<Rectangle>() ;
        recPair.setFirst(new Rectangle(0, 0, 10, 10)) ;
        recPair.setSecond(new Rectangle(0, 0, 100, 100));
        System.out.println( recPair.getFirst() + " " + recPair.getSecond()) ;
    }
}
```

Generic Methods

- You can define generic methods inside an ordinary class.

```
class ArrayAlg {
  public static <T> T getMiddle( T[] a) {
    return a[a.length/2]) ;
  }
}
```

The type variable T is inserted between the modifiers and the return type

- When you call a generic method, you can place the actual type before the method name.

```
String [] names = {“John”, “Q”, “Public”} ;
String middle = ArrayAlg.<String>getMiddle(names) ;
// simplely, when the actual type can be inferred
String middle = ArrayAlg.getMiddle(names) ;
```

Bounds for Type Variable

```
class ArrayAlgForString { // Not generic. It is only for String
   public static Pair<String> minmax(String[] a)  {
      String min = a[0], max = a[0];
      for (int i = 1; i < a.length; i++) {
         if (min.compareTo(a[i]) > 0) min = a[i];
         if (max.compareTo(a[i]) < 0) max = a[i];
      }
      return new Pair<String>(min, max);
   }
}

public class PairTest2 {
   public static void main(String[] args) {
      String[] words = { "cd", "ab", "lm", "ef" };
      Pair<String> mm = ArrayAlgForString.minmax(words);
      System.out.println("min = " + mm.getFirst());
      System.out.println("max = " + mm.getSecond());
   }
}
```

Bounds for Type Variable

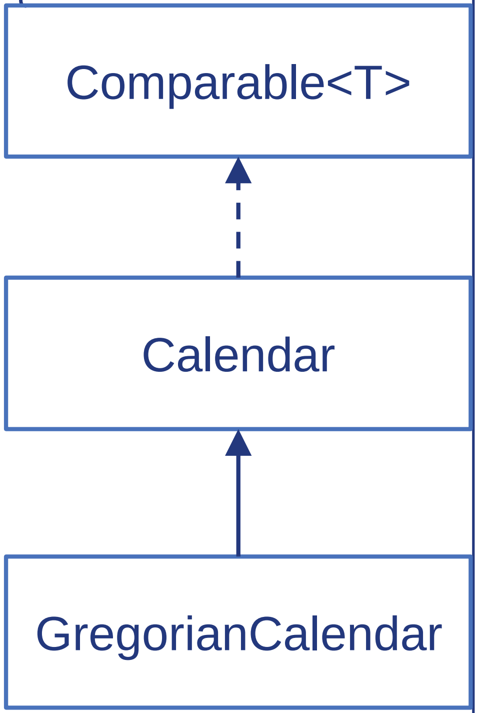
```
class ArrayAlg {
  // interface java.lang.Comparable<T>
  // int compareTo(T object)
  public static <T extends Comparable<T>> Pair<T> minmax(T[] a)  {
    T min = a[0], max = a[0];
    for (int i = 1; i < a.length; i++) {
      if (min.compareTo(a[i]) > 0) min = a[i];
      if (max.compareTo(a[i]) < 0) max = a[i];
    }
    return new Pair<T>(min, max);
  }
}
public class PairTest3 {
  public static void main(String[] args) {
    String[] words = { "cd", "ab", "lm", "ef" };
    Pair<String> mm = ArrayAlg.minmax(words);
    System.out.println("min = " + mm1.getFirst() + " max = " + mm1.getSecond());

    Rectangle[] rectangles = { new Rectangle(0, 0, 10, 10), new Rectangle(0, 0, 20, 20) };
    Pair<Rectangle> mm2 = ArrayAlg.minmax(rectangles);
    System.out.println("min = " + mm2.getFirst() + " max = " + mm2.getSecond());
  }
}
```

T is guaranteed to provide compareTo() because it implements Comparable<T>

The method minmax(T[]) in the type ArrayAlg is not applicable for the arguments (Rectangle[])

```
import java.util.*;
class ArrayAlg {
public static <T extends Comparable<T>> Pair<T> minmax(T[] a) {
      T min = a[0];
       T max = a[0];
       for (int i = 1; i < a.length; i++) {
           if (min.compareTo(a[i]) > 0) min = a[i];
           if (max.compareTo(a[i]) < 0) max = a[i];
       }
       return new Pair<T>(min, max);
   }
}
public class PairTest4 {
   public static void main(String[] args) {
       Calendar[] birthdays = {
              // java.util.GregorianCalendar extends java.util.Calendar
              // java.util.Calendar implements Comparable<Calendar>
             new GregorianCalendar(1906, Calendar.DECEMBER, 9),
             new GregorianCalendar(1815, Calendar.DECEMBER, 10),
             new GregorianCalendar(1903, Calendar.DECEMBER, 3),
             new GregorianCalendar(1910, Calendar.JUNE, 22)
          };
       Pair<Calendar> mm = ArrayAlg.minmax(birthdays);
       System.out.println("min = " + mm.getFirst().getTime());
       System.out.println("max = " + mm.getSecond().getTime());
   }
}
```



No problem !
Because Calendar implements Comparable<Calendar>

INNER CLASS

Inner Classes

- An inner class is a class that is defined inside another class

```
class OuterClass {
     …
     private class InnerClass {
          …
     }
}
```

Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.

inner class can be hidden from the outside world.

- Three kinds of inner classes
  - Ordinary Inner class
  - Local (inner) class
  - Anonymous (inner) class

```
public class IntArray {
    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];

    public IntArray() {
        for (int i = 0; i < SIZE; i++) arrayOfInts[i] = i;
    }
    public void printEven() { // print out values of even indices of the array
        InnerEvenIterator iterator = this.new InnerEvenIterator();
        while (iterator.hasNext())
            System.out.println(iterator.getNext() + " ");
    }
    // inner class implements the Iterator pattern
    private class InnerEvenIterator {
        // start stepping through the array from the beginning
        private int next = 0;
        public boolean hasNext() { return next <= SIZE – 1; }
        public int getNext() {
            final int retValue = arrayOfInts[next];
            next += 2;
            return retValue;
        }
    }
    public static void main(String s[]) {
        // fill the array with integer values and print out only values of even indices
        IntArray ia = new IntArray();
        ia.printEven(); // 0 2 4 6 8 10 12 14
    }
}
```

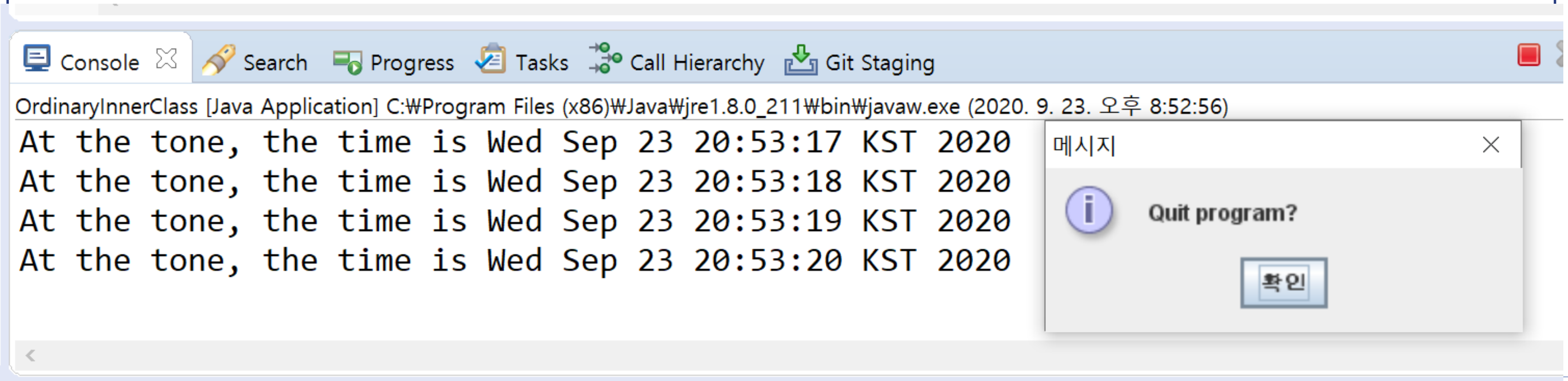
An instance of InnerClass exist only within an instance of OuterClass

An instance of InnerClass has direct access to the methods and fields of its enclosing instance.

Inner Class: An Example

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

public class OrdinaryInnerClass {
   public static void main(String[] args) {
      TalkingClock clock = new TalkingClock (1000, true);
      clock.start();
      // keep program running until user selects "Ok"
      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}
```



OrdinaryInnerClass [Java Application] C:₩Program Files (x86)₩Java₩jre1.8.0_211₩bin₩javaw.exe (2020. 9. 23. 오후 8:52:56)

```
At the tone, the time is Wed Sep 23 20:53:17 KST 2020
At the tone, the time is Wed Sep 23 20:53:18 KST 2020
At the tone, the time is Wed Sep 23 20:53:19 KST 2020
At the tone, the time is Wed Sep 23 20:53:20 KST 2020
```

메시지

Quit program?

```
class TalkingClock { // non-public class
    public TalkingClock(int interval, boolean beep) {  this.interval = interval; this.beep = beep; }
    public void start() {
        ActionListener listener = new TimePrinter(); // create TimePrinter Object
        Timer t = new Timer(interval, listener);
      // javax.swing.Timer(int delay, ActionListener listener)
        t.start();
    }
    private int interval;
    private boolean beep;

    // ordinary inner class
    private class TimePrinter implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            Date now = new Date();
            System.out.println("At the tone, the time is " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }
}
```

An instance of InnerClass exist only within an instance of OuterClass

TimePrinter is defined inside TalkingClock

TimePrinter cannot be used outsideTalkingClock

An instance of InnerClass has direct access to the methods and fields of its enclosing instance.
beep is equivalent to **TalkingClock.this.beep**

ActionListener Interface

- The timer needs to know what method to call.
- The timer requires that you specify an object of a class that implements the **ActionListener** interface of the java.awt.event package.

```
public interface ActionListener {
    void actionPerformed(ActionEvent event);
}
```

- The timer calls the **actionPerformed** method when the time interval has expired

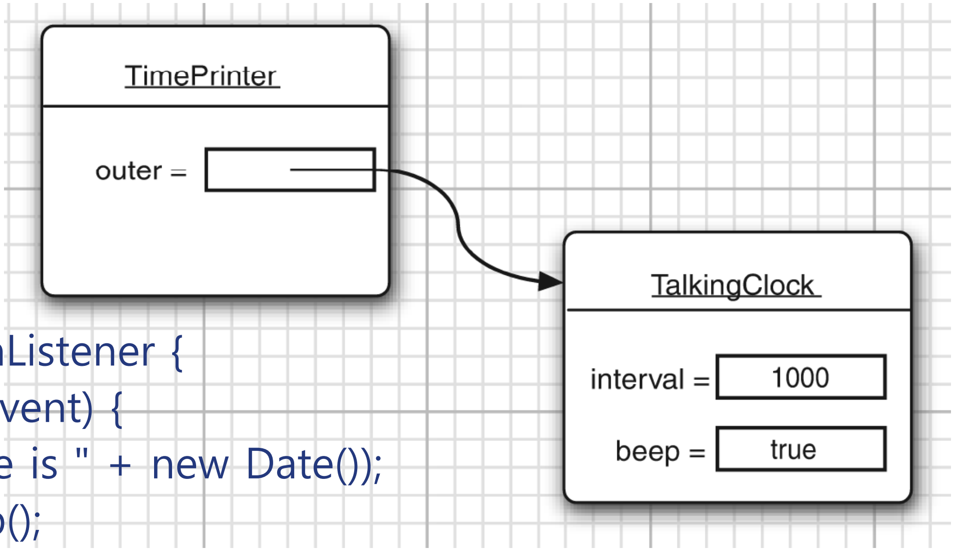
```
private class TimePrinter implements ActionListener {
        public void actionPerformed(ActionEvent event) {
           Date now = new Date();
           System.out.println("At the tone, the time is " + now);
           if (beep) Toolkit.getDefaultToolkit().beep();
        }
   }
```

Inner Class

- An inner class object has a reference to an outer class object

```
class TalkingClock { // outer class
   public TalkingClock(int interval, boolean beep) {  this.interval = interval; this.beep = beep; }
   public void start() {
      …
   }
   private int interval;
   private boolean beep;

   // ordinary inner class
   private class TimePrinter implements ActionListener {
      public void actionPerformed(ActionEvent event) {
         System.out.println("At the tone, the time is " + new Date());
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }
}
```



Local Inner Classes

- You can define a class locally **inside a single method.**
- A local inner class can access the fields of their outer classes.

```
class TalkingClock {
  public void start(int interval, final boolean beep) {
      class TimePrinter implements ActionListener {  // local inner class
         public void actionPerformed(ActionEvent event) {
            Date now = new Date();
            System.out.println("At the tone, the time is " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
         }
      }
      ActionListener listener = new TimePrinter();
      Timer t = new Timer(interval, listener);
      t.start();
   }
//   private int interval;
//   private boolean beep;
}
```

Local Inner Classes

- In addition, a local inner class can access local variables, but they must be final.

```
class TalkingClock {
  public void start(int interval, final boolean beep) {
      class TimePrinter implements ActionListener {
         public void actionPerformed(ActionEvent event) {
            Date now = new Date();
            System.out.println("At the tone, the time is " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
         }
      }
      ActionListener listener = new TimePrinter();
      Timer t = new Timer(interval, listener);
      t.start();
   }
}
```

Anonymous Inner Classes

- You can define a local inner class without name

```
class TalkingClock {
  public void start(int interval, final boolean beep) {
    ActionListener listener = new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        Date now = new Date();
        System.out.println("At the tone, the time is " + now);
        if (beep) Toolkit.getDefaultToolkit().beep();
      }
    } ;
    Timer t = new Timer(inverval, listener);
    t.start();
  }
}
```

Create a new object of a class that implements the ActionListener interface

Event Handling with Lambda Expression

- The implementation of a **single method interface** ActionListener interface is specified by a lambda expression

```
class TalkingClock {
  public void start(int interval, final boolean beep) {
     Timer t = new Timer(1000,
        (ActionEvent event) -> {
            Date now = new Date();
            System.out.println("At the tone, the time is " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
     );
     t.start();
  }
}
```

Implementation of actionPerformed() of interface ActionListener

Q&A