Generic classes

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.
- Support Raw type, but erasure of generic types and methods

raw type

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

Type parameter
(Type variable)

parameterized type

```
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 Class: Bounded Type Parameters

```
class Pair<T extends Number & Serializable> {
    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 PairTest2 {
    public static void main(String[] args) {
        Pair<Integer> intPair = new Pair<Integer>() ; // Integer is parameterized type
        intPair.setFirst(1) ;
        intPair.setSecond(100);
        System.out.println( intPair.getFirst() + " " + intPair.getSecond()) ;

        Pair<Float> floatPair = new Pair<Float>() ;
        floatPair.setFirst(1.1F) ;
        floatPair.setSecond(100.1F);
        System.out.println( floatPair.getFirst() + " " + floatPair.getSecond()) ;
    }
}
```

Restrictions on Generics (1/2)

- Cannot Instantiate Generic Types with Primitive Types

```
Pair<int, char> p = new Pair<>(8, 'a');  //  error
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

- Cannot Create Instances of Type Parameters

```
public static <E> void append(List<E> list) {
    E elem = new E();  // error
    list.add(elem);
}
```

- Cannot Use Casts or instanceof with Parameterized Types

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // error
        // ...
    }
}
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // error
```

https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html

Restrictions on Generics (2/2)

- Cannot Declare Static Fields Whose Types are Type Parameters

```
class Pair<T> {
        static T first; //error
        static Pair<T> minmax (T[] a) { }  //error
        …
}
```

- Cannot Create Arrays of Parameterized Types

```
class Pair<T> {
        T[] items;    // ok
        T[] toArray() {
             T[] tmpArr = new T[items.length];  //error
              …
              return tmpArr;
         }
}
```

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 parameter 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 middle1 = ArrayAlg.<String>getMiddle(names) ;
// simplely, when the actual type can be inferred
String middle2 = ArrayAlg.getMiddle(names) ;
```

Bounds for Type Parameter

```
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 PairTest3 {
    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 Parameter

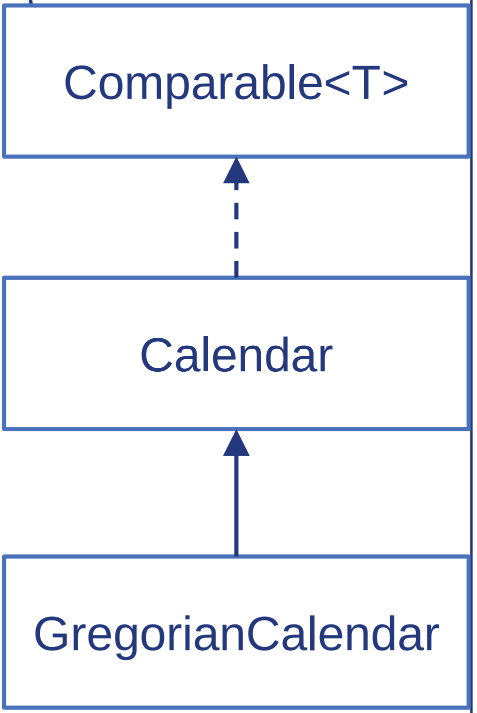
```
class ArrayAlg {
  // interface java.lang.Comparable<T>
  // int compareTo(T object)
  public static <T extends Comparable<T>> Pair<T> minmax (T[] a)  {
    T min = a[0], max = a[0];
    for (int i = 1; i < a.length; i++) {
      if (min.compareTo(a[i]) > 0) min = a[i];
      if (max.compareTo(a[i]) < 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);  // type inference
    System.out.println("min = " + mm1.getFirst() + " max = " + mm1.getSecond());

    // Rectangle does not implement Comparable interface.
    Rectangle[] rectangles = { new Rectangle(0, 0, 10, 10), new Rectangle(0, 0, 20, 20) };
    //Pair<Rectangle> mm2 = ArrayAlg.<Rectangle>minmax(rectangles); //compile error
    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>

Polymorphism in Generic Type

- the generic type in the reference variable and the generic type in the new operator **must match**.

```
Pair <String> pair = new Pair<String>();
List<Integer> list1 = new ArrayList<Integer>();
```

```
// Pair <Number> pair = new Pair<Integer>();
// List<Number> list2 = new ArrayList<Integer>();
```

- Wild Card
  - it may be only used as reference parameters
  - <? extends T>: T and it's subclass
  - <? super T>   : T and it's superclass
  - <?>   : all classes (unbounded)

```
Pair <? extends Number> pair = new Pair<Integer>();
List <? extends Number> list2 = new ArrayList<Integer>();
```

```
void printCollection(Collection<Object> c) {
    for (Object e : c)
        System.out.println(e);
}
void printCollection(Collection<?> c) {
    for (Object e : c)
        System.out.println(e);
}
```

```
public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes)
        s.draw(this);
}
public void drawAll(List<? extends Shape> shapes) {
    for (Shape s: shapes)
        s.draw(this);
}
```

https://docs.oracle.com/javase/tutorial/extra/generics/wildcards.html

Sorting with Comparable<T> and Comparator<T>

Sorting Array of Basic Types

```
public class BasicSortingMain {
  public static void main(String[] args) {
       int[] intArr = {5,9,1,10};
       Arrays.sort(intArr);
       System.out.println(Arrays.toString(intArr));

       String[] strArr = {"A", "C", "B", "Z", "E"};
       Arrays.sort(strArr);
       System.out.println(Arrays.toString(strArr));

       List<String> strList = new ArrayList<>();
       strList.add("A");
       strList.add("C");
       strList.add("B");
       strList.add("Z");
       strList.add("E");
       Collections.sort(strList);
       for ( String str: strList ) System.out.print(" "+str);
  }
}
```

```
[1, 5, 9, 10]
[A, B, C, E, Z]
 A B C E Z
```

Class Employee

- To sort an Object by its property, you have to make the Object implement the Comparable interface and override the compareTo() method
- A negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
public class Employee implements Comparable<Employee> {
   private int id;
   private String name;
   private int age;
   private long salary;

   public Employee(int id, String name, int age, int salary) {
       // constructor
   }

   @Override
   public int compareTo(Employee emp) {
     return (this.id - emp.id);
   }
}
```

Sorting with Comparable<T> Interface

```
public class SortingObjectMain {
  public static void main(String[] args) {
    //sorting object array
    Employee[] empArr = new Employee[4];
    empArr[0] = new Employee(10, "Mikey", 25, 10000);
    empArr[1] = new Employee(20, "Arun", 29, 20000);
    empArr[2] = new Employee(5, "Lisa", 35, 5000);
    empArr[3] = new Employee(1, "Pankaj", 32, 50000);

    // Employee should implement Comparable<Employee> interface
    Arrays.sort(empArr);

    System.out.println("Default Sorting of Employees list:₩n
        +Arrays.toString(empArr));
  }
}
```

```
Default Sorting of Employees list:
[[id=1, name=Pankaj, age=32, salary=50000], [id=5, name=Lisa, age=35, salary=5000],
[id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000]]
```

Class Employee having Comparators

- The Comparable interface is only allow to sort a single property. To sort with multiple properties, you need **Comparator<T>**.

```
public class Employee implements Comparable<Employee> {
   ...
   public int compareTo(Employee emp) { return (this.id - emp.id); }

   public static Comparator<Employee> SalaryComparator
      = new Comparator<Employee>() {
          public int compare(Employee e1, Employee e2) {
              return (int) (e1.getSalary() - e2.getSalary());
          }
     };
   public static Comparator<Employee> AgeComparator
      = new Comparator<Employee>() {
          public int compare(Employee e1, Employee e2) {
              return e1.getAge() - e2.getAge();
          }
     };
   public static Comparator<Employee> NameComparator
      = new Comparator<Employee>() {
          public int compare(Employee e1, Employee e2) {
              return e1.getName().compareTo(e2.getName());
          }
     };
}
```

Sorting with Comparator<T> Interface

```
public class SortingObjectMain {
   public static void main(String[] args) {
      //sorting object array
      Employee[] empArr = new Employee[4];
      …

      //sort employees array using Comparator by Salary
      Arrays.sort(empArr, Employee.SalaryComparator);
      System.out.println("Employees list sorted by Salary:₩n“
           +Arrays.toString(empArr));

      //sort employees array using Comparator by Age
      Arrays.sort(empArr, Employee.AgeComparator);
      System.out.println("Employees list sorted by Age:₩n“
           +Arrays.toString(empArr));

      //sort employees array using Comparator by Name
      Arrays.sort(empArr, Employee.NameComparator);
      System.out.println("Employees list sorted by Name:₩n“
           +Arrays.toString(empArr));
   }
}
```

```
Employees list sorted by Salary:
[[id=5, name=Lisa, age=35,
salary=5000], [id=10, name=Mikey,
age=25, salary=10000], [id=20,
name=Arun, age=29, salary=20000],
[id=1, name=Pankaj, age=32,
salary=50000]]
Employees list sorted by Age:
[[id=10, name=Mikey, age=25,
salary=10000], [id=20, name=Arun,
age=29, salary=20000], [id=1,
name=Pankaj, age=32, salary=50000],
[id=5, name=Lisa, age=35, salary=5000]]
Employees list sorted by Name:
[[id=20, name=Arun, age=29,
salary=20000], [id=5, name=Lisa,
age=35, salary=5000], [id=10,
name=Mikey, age=25, salary=10000],
[id=1, name=Pankaj, age=32,
salary=50000]]
```

Q&A