Data Structures for Java

William H. Ford
William R. Topp

Chapter 5

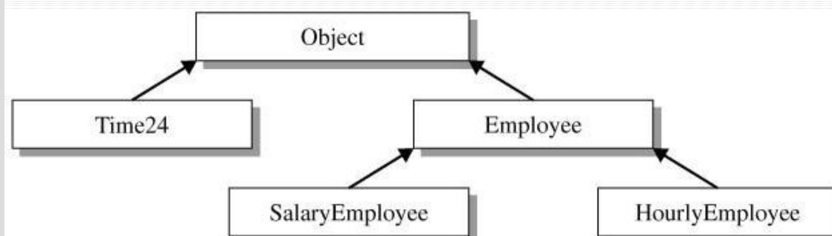
Generic Classes and Methods

Bret Ford

© 2005, Prentice Hall

The Object Superclass

- The Object class defines a variety of methods, some of which have applications in operating systems and systems programming.



Object Class Methods

- Boolean equals(Object obj)
  Indicates whether some other object is "equal to" this one. A class implements the method by first checking whether obj is an instance of the class.
- String toString()
  Returns a string representation of the object

Time24 equals()

```
public boolean equals(Object item)
{
   // check the obj is a Time24 object
   if (item instanceof Time24)
   {
      // convert objects to minutes
      // and compare as integers
      Time24 t = (Time24)item;
      int time, ttime;
      time = hour * 60 + minute;
      ttime = t.hour * 60 + t.minute;
      // compare the two times
      return time == ttime;
   }
   // argument is not a Time24 object
   return false;
}
```

Generalized toString(arr)

```
// returns a string that represents
// an array of objects; found in Arrays class
public static String toString(Object[] arr)
{
   if (arr == null)
      return "null";
   else if (arr.length == 0)
      return "[]";
   // start with the left bracket
   String str = "[" + arr[0];
   // output all but the last element,
   // separating items with a comma.
   for (int i = 1; i < arr.length; i++)
      str += ", " + arr[i];
   str += "]";
   return str;
}
```

Generalized Sequential Search

- A generalized version of the sequential search specifies an array and a target of type Object. The array type must override the Object equals() method. The scan of the array uses the equals() method and polymorphism to locate the index of the target value.

Sequential Search with int

```
public static int seqSearch(int[] arr, int first,
                            int last, int target)
{
   // scan elements in the range
   // first <= i < last; test for a match
   for (int i = first; i < last; i++)
      if (arr[i] == target)
         return i;

   // match not found
   return -1;
}
```

Sequential Search with Object

```
public static int seqSearch(Object[] arr,
                            int first, int last,
                            Object target)
{
   // scan elements in the range
   // first <= i < last; test for a match
   for (int i = first; i < last; i++)
      if (arr[i].equals(target))
         return i;

   // match not found
   return -1;
}
```

Generalized Collections with Object References

- A generalized collection class stores elements of different object types. One technique stores elements of universal Object type.
- Issues:
  - A cast is required to access any element in the collection
  - Objects of different types can be added to the collection. Access (with a cast) can result in a runtime error.

Object-based Store Class

```
public class Store
{
    // data stored by the collection
    private Object value;
    // constructor creates object with initial value v
    public Store (Object v)
    { value = v; }
    // return the stored value
    public Object getValue()
    { return value; }
    // set v as the new stored value
    public void setValue(Object v)
    { value = v; }
    // a description of the element
    public String toString()
    { return "value = " + value; }
}
```

Generalized Collections with Generics

- With a generic collection class, instances are created with a specific object type.
- A generic collection class is "type-safe" meaning that the compiler identifies statements that incorrectly insert or access an element.
- No cast is required when accessing an element.

Creating and Using Generic Collections

- Declare a generic collection class by following the class name is a generic type in angle brackets "<T>".
  E.g. public class GenClass<T>
- Create a generic class object by including the type in angle bracket ("<type>" after the class name.
  E.g. GenClass<String> strCollection;

Genetic Store Collection

- Store<T> is a prototype of a generic class. Its definition includes:
  - Header with generic type <T>
  - Generic instance variable `value`
  - Access method `getValue()` with a generic return type
  - Update method `setValue(T v)` with a generic type parameter

Generic Store Class

```
public class Store<T>
{
    // data stored by the object
    private T value;
    // constructor creates an object
    // with initial value
    public Store (T v)
    { value = v; }
    // return the stored value as type T
    public T getValue()
    { return value; }
    // update the stored value
    public void setValue(T v)
    { value = v; }
    public String toString()
    { return "Value = " + value; }
}
```

Generic Interface

- Interfaces can be generic. Use a generic type in the header and in the methods. Supply the actual type when declaring a class that implements the interface.
  E.g. Accumulator<T> interface

```
public interface Accumulator<T>
{
    public void add(T v);
}
```

Declaring class with interface of type String.

```
class Demo implements Accumulator<String>
```

AccumulatorTime Class

```
public class AccumulatorTime implements
             Accumulator<Time24>
{
   private Time24 total;

   // constructor creates an object
   // with initial time 0:00
   public AccumulatorTime ()
   { total = new Time24(); }
   // return the total
   public Time24 getTotal()
   { return total; }
   // update the total time by the
   // specified Time24 amount v
   public void add(Time24 v)
   { total.addTime(v.getHour()*60 + v.getMinute()); }
}
```

AccumulatorNumber Class

```
public class AccumulatorNumber implements
             Accumulator<Number>
{
   // accumulation of numeric values
   private double total;

   // constructor initializes total to 0.0
   public AccumulatorNumber ()
   { total = 0.0; }
   // return the total
   public double getTotal()
   { return total; }
   // update the total by the value of v as a double
   public void add(Number v)
   { total = total + v.doubleValue(); }
}
```

Program 5.1

```
import ds.time.Time24;

public class Program5_1
{
   public static void main (String[] args)
   {
      Integer[] intArr = {7, 1, 9, 3, 8, 4};
      String[] strArr = {"3:45", "2:30", "5:00"};

      AccumulatorNumber accNumber =
                  new AccumulatorNumber();
      AccumulatorTime accTime =
                  new AccumulatorTime();
```

Program 5.1 concluded

```
        int i;
        for (i = 0; i < intArr.length; i++)
           accNumber.add(intArr[i]);
        System.out.println("Numeric total is " +
                           accNumber.getTotal());

        for (i = 0; i < strArr.length; i++)
           accTime.add(Time24.parseTime(strArr[i]));
        System.out.println("Time total is " +
                           accTime.getTotal());
    }
}
```

```
Run:

Numeric total is 52.4
Time total is 11:15
```

Comparable Interface

- The Comparable<T> inteface defines a standard way to compare objects using relations less than, equal to, and greater than.
- The interface defines a single method compareTo(T item)

```
public interface Comparable<T>
{
   int compareTo(T item);
}
```

Interpreting compareTo()

- The method compareTo() returns an integer value that is negative, zero, or positive. The value compares the value of an attributes of the object (obj) with another object (item) of the same type
- obj.compareTo(item) < 0 obj < item
- obj.compareTo(item) == 0 obj == item
- obj.compareTo(item) > 0 obj <> item

Time24 Class with Comparable

```
public class Time24 implements Comparable<Time24>
{   . . .
    // compareTo() compares times converted to minutes
    public int compareTo(Time24 item)
    {
        int time, ttime;
        time = hour * 60 + minute;
        ttime = item.hour * 60 + item.minute;
        // compare the integer values and return -1, 0, or 1
        if (time < ttime)
           return -1;
        else if (time == ttime)
           return 0;
        else
           return 1;
    }
}
```

Generic Methods

- Methods can be generic. They are used to implement generic algorithms.

  The method header includes the generic type <T> as one of the modifiers and must contain a generic parameter.

  E.g. public static <T> T max(T objA, T objB)

Generic max() (basic generic definition)

The compiler issues an "unchecked cast warning. This form of the method is NOT "type-safe".

```
public static <T> T max(T objA, T objB)
{
    if (((Comparable<T>)objA).compareTo(objB) > 0)
        return objA;
    else
        return objB;
}
```

Generic max() with bounded type

A bound specifies that generic type T must implement the Comparable<T> interface. Failing to do so results in a compiler error message. Use the reserved word "extends" in the generic tag to indicate the type implements Comparable<T>.

```
public static <T extends Comparable<T>> T max(T objA, T objB)
{
   if (objA.compareTo(objB) > 0)
      return objA;
   else
      return objB;
}
```

Generic max() with a Wildcard

The syntax <?> indicates that the generic structure is handling an unknown type. A generic method combines the "?" tag with the "super" bound to indicate that the type or some superclass of the type implements the Comparable interface.

```
public static <T extends Comparable<? super T>>
     T max(T a, T b)
{
    if (a.compareTo(b) > 0)
       return a;
    else
       return b;
}
```

Generic Searching/Sorting

- A generic sort methods use an array of specified type. A generic search method also has a target parameter of the specified type.
- If the method uses compareTo() to compare the relative value of objects, use the generic tag
- <T extends Comparable<? super T>>

Generic Selection Sort

```
public static <T extends Comparable<? super T>>
       void selectionSort(T[] arr)
{
   // index of smallest element in the sublist
   int smallIndex;
   int pass, j, n = arr.length;
   T temp;

   // pass has the range 0 to n-2
   for (pass = 0; pass < n-1; pass++)
   {
      // scan the sublist starting at index pass
      smallIndex = pass;
```

Generic Selection Sort (2)

```
      // j traverses the sublist arr[pass+1] to arr[n-1]
      for (j = pass+1; j < n; j++)
      // if smaller element found, assign smallIndex
      // to that position
      if (arr[j].compareTo(arr[smallIndex]) < 0)
         smallIndex = j;

      // swap the next smallest element into arr[pass]
      temp = arr[pass];
      arr[pass] = arr[smallIndex];
      arr[smallIndex] = temp;
   }
}
```

Generic binSearch()

```
public static <T extends Comparable<? super T>>
int binSearch(T[] arr, int first, int last, T target)
{
   // index of the midpoint
   int mid;
   // object that is assigned arr[mid]
   T midvalue;
   // save original value of last
   int origLast = last;

   // test for nonempty sublist
   while (first < last)
   {
      mid = (first+last)/2;
      midvalue = arr[mid];
```

Generic binSearch() (2)

```
        if (target.compareTo(midvalue) == 0)
            // have a match
            return mid;
        // determine which sublist to search
        else if (target.compareTo(midvalue) < 0)
            // search lower sublist. reset last
            last = mid;
        else
            // search upper sublist. reset first
            first = mid+1;
    }
    // target not found
    return -1;
}
```

Program 5.2

```
import ds.util.Arrays;
import ds.time.Time24;

public class Program5_2
{
   public static void main (String[] args)
   {
      String[] strArr = {"red", "green", "blue"};
      Integer[] intArr = {40, 70, 50, 30};
      Time24[] timeArr = {new Time24(14,15),
                          new Time24(10, 45),
                          new Time24(22,00),
                          new Time24(3,30)};
      SalaryEmployee[] emp = {
       new SalaryEmployee("Dunn, Moira", "471-23-8092",800),
       new SalaryEmployee("Garcia, Avey","398-67-1298",1200),
       new SalaryEmployee("Ye, Don","682-76-1298",2000)};
```

Program 5.2 (concluded)

```
        Arrays.selectionSort(strArr);
        System.out.println("Sorted strings: " +
                           Arrays.toString(strArr));
        Arrays.selectionSort(intArr);
        System.out.println("Sorted integers: " +
                           Arrays.toString(intArr));
        Arrays.selectionSort(timeArr);
        System.out.println("Sorted times: " +
                           Arrays.toString(timeArr));
        Arrays.selectionSort(emp);
        for (int i=0; i < emp.length; i++)
           System.out.println(emp[i].payrollCheck());
    }
}
```

Program 5.2 Run

```
Run:

Sorted strings: [blue, green, red]
Sorted integers: [30, 40, 50, 70]
Sorted times: [3:30, 10:45, 14:15, 22:00]
Pay Garcia, Avey (398-67-1298) $1200.00
Pay Dunn, Moira (471-23-8092) $800.00
Pay Ye, Don (682-76-1298) $2000.00
```