Supplemental Resource Materials for

Data Structures and Algorithms in Java

Third Edition

Michael T. Goodrich
Department of Computer Science
Johns Hopkins University

Roberto Tamassia
Department of Computer Science
Brown University

John Wiley & Sons, Inc.
New York • Chichester • Weinheim • Brisbane • Singapore • Toronto

Contents

Appendix A Supplemental Resource Materials

Contents

This document contains supplemental material for the book *Data Structures and Algorithms in Java*, third edition (DSAJ3), written by Michael T. Goodrich and Roberto Tamassia, which is published by John Wiley & Sons, Inc. in 2004. Each of the sections of this document make use of material from DSAJ3, but are otherwise completely self contained.

In particular, this supplement contiains concise discussions on new features added to the Java language as a part of the Java 5.0 release. These sections will be included in the fourth edition of *Data Structures and Algorithms in Java*, by Goodrich and Tamassia.

A.1 Enum Types

Since 5.0, Java supports enumerated types, called ***enums***. These are types that are allowed only to take on values that come from a specified set of names. They are declared inside of a class as follows:

modifier **enum** *name* { $value\_name_0$, $value\_name_1$, ..., $value\_name_{n-1}$ };

where the *modifier* can be blank, **public**, **protected**, or **private**. The name of this enum, *name*, can be any legal Java identifier. Each of the value identifiers, $value\_name_i$, is the name of a possible value that variables of this enum type can take on. These name values can each also be any legal Java identifier, but the Java convention is to use capitalized words. For example, the following enumerated type definition might be useful in a program that must deal with dates:

public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };

Once defined, we can use an enum type, such as this, to define other variables, much like a class name. But since Java knows all the value names for an enumerated type, if we use an enum type in a string expression, Java will automatically use its name. Enum types also have a few built-in methods, including a method valueOf, which returns the enum value that is the same as a given string. We show an example use of an enum type in Code Fragment A.1.

```
public class DayTripper {
  public enum Day {MON, TUE, WED, THU, FRI, SAT, SUN};
  public static void main(String[] args) {
    Day d = Day.MON;
    System.out.println("Initially d is " + d);
    d = Day.WED;
    System.out.println("Then it is " + d);
    Day t = Day.valueOf("WED");
    System.out.println("I say d and t are the same: " + (d == t));
  }
}
```

The output from this program is:

```
Inititally d is MON
Then it is WED
I say d and t are the same: true
```

Code Fragment A.1: An example use of an enum type.

A.2 Simple Input Using the java.util.Scanner Class

Just as there is a special object for performing output to the Java console window, there is also a special object, called System.in, for performing input from the Java console window. Technically, the input is actually coming from the "standard input" device, which by default is the computer keyboard echoing its characters in the Java console. The System.in object is an object associated with the standard input device. A simple way of reading input with this object is to use it to create a Scanner object, using the expression

```
new Scanner(System.in)
```

The Scanner class includes a number of convenient methods that read from the given input stream. For example, the following program uses a Scanner object to process input:

```
import java.io.*;
import java.util.Scanner;
public class InputExample {
  public static void main(String args[]) throws IOException {
    Scanner s = new Scanner(System.in);
    System.out.print("Enter your height in centimeters: ");
    float height = s.nextFloat();
    System.out.print("Enter your weight in kilograms: ");
    float weight = s.nextFloat();
    float bmi = weight/(height*height)*10000;
    System.out.println("Your body mass index is " + bmi + ".");
  }
}
```

When executed, this program could produce the following on the Java console:

```
Enter your height in centimeters: 180
Enter your weight in kilograms: 80.5
Your body mass index is 24.84568.
```

java.util.Scanner Methods

The Scanner class reads the input stream and divides it into ***tokens***, which are contiguous strings of characters separated by ***delimiters***, which are special separating characters. The default delimiter is whitespace; that is, tokens are are separated by strings of spaces, tabs, and newlines, by default. Tokens can either be read immediately as strings or a Scanner object can converted a token to a base type, if the token is in the right syntax. Specifically, the Scanner class includes the following methods for dealing with tokens:

hasNext(): Returns **true** if and only if there is another token in the input stream.

next(): Returns the next token string in the input stream; it generates an error if there are no more tokens left.

hasNext*Type*(): Returns **true** if and only if there is another token in the input stream and it can be interpreted as the corresponding base type, *Type*, where *Type* can be Boolean, Byte, Double, Float, Int, Long, or Short.

next*Type*(): Returns the next token in the input stream, returned as the base type corresponding to *Type*; it generates an error if there are no more tokens left or if the next token cannot be interpreted as a base type corresponding to *Type*.

Additionally, Scanner objects can process input line-by-line, ignoring delimiters, and even look for patterns within lines while doing so. The methods for processing input in this way include the following:

hasNextLine(): Returns **true** if and only if the input stream has another line of text.

nextLine(): Advances the input past the current line ending and returns the input that was skipped.

findInLine(String s): Attempts to find a string matching the (regular expression) pattern s in the current line. If the pattern is found, it is returned and the scanner advances to the first character after this match. If the pattern is not found, the scanner returns **null** and doesn't advance.

These methods can be used with those above, as well, as in the following:

```
Scanner input = new Scanner(System.in);
System.out.print("Please enter an integer: ");
while (!input.hasNextInt()) {
  input.nextLine();
  System.out.print("That's not an integer; please enter an integer: ");
}
int i = input.nextInt();
```

A.3 Casting and Autoboxing/Unboxing

Casting is an operation that allows us to change the type of the value returned by an expression. In its simplest application, we can take a variable of one type and ***cast*** it into an equivalent variable of another type. Casting can be useful for doing certain numerical and input/output operations.

The syntax for casting an expression to a desired type is as follows:

(*type*) *exp*

where *type* is the type that we would like the expression *exp* to have. There are two fundamental types of casting that can be done in Java. We can either cast with respect to the base numerical types or we can cast with respect to objects. For instance, it might be helpful to cast an **int** to a **double** in order to perform operations like division.

Ordinary Casting

When casting from a **double** to an **int**, we may lose precision. This means that the resulting double value will be rounded down. But we can cast an **int** to a **double** without this worry. For example, consider the following:

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int)d1;          // i1 has value 3
int i2 = (int)d2;          // i2 has value 3
double d3 = (double)i2;    // d3 has value 3.0
```

Casting with Operators

Certain binary operators, like division, will have different results depending on the variable types they are used with. We must take care to make sure operations perform their computations on values of the intended type. When used with integers, division does not keep track of the fractional part, for example. When used with doubles, division keeps this part, as is illustrated in the following example:

```
int i1 = 3;
int i2 = 6;
dresult = (double)i1 / (double)i2;  // dresult has value 0.5
dresult = i1 / i2;                  // dresult has value 0.0
```

Notice that when i1 and i2 were cast to doubles, regular division for real numbers was performed. When i1 and i2 were not cast, the " /" operator performed an integer division and the result of i1 / i2 was the **int** 0. Java then did an ***implicit cast*** to assign an **int** value to the **double** result. We discuss implicit casting next.

Implicit Casting and Autoboxing/Unboxing

There are cases where Java will perform an ***implicit cast***, according to the type of the assignment variable, provided there is no loss of precision. For example:

```
int iresult, i = 3;
double dresult, d = 3.2;
dresult = i / d;          // dresult is 0.9375. i was cast to a double
iresult = i / d;          // loss of precision -> this is a compilation error
iresult = (int) i / d;    // iresult is 0, since the fractional part is lost
```

Note that since Java will not perform implicit casts where precision is lost, the explicit cast in the last line above is required.

Since Java 5.0, there is a new kind of implicit cast, for going between Number objects, like Integer and Float, and their related base types, like **int** and **float**. Any time a Number object is expected as a parameter to a method, the corresponding base type can be passed. In this case, Java will perform an implicit cast, called ***autoboxing***, which will convert the base type to its corresponding Number object. Likewise, any time a base type is expected in an expression involving a Number reference, that Number object is changed to the corresponding base type, in an operation called ***unboxing***.

There are few caveats regarding autoboxing and unboxing, however. One is that if a Number reference is **null**, then trying to unbox it will generate an error, called **NullPointerException**. Second, the operator, "==", is used both to test the equality of base type values as well as whether two Number references are pointing to the same object. So when testing for equality, try to avoid the implicit casts done by autoboxing and unboxing. Finally, implicit casts, of any kind, take time; so we should try to minimize our reliance on them if performance is an issue.

Incidentally, there is one situation in Java when only implicit casting is allowed, and that is in string concatenation. Any time a string is concatenated with any object or base type, that object or base type is automatically converted to a string. Explicit casting of an object or base type to a string is not allowed, however. Thus, the following assignments are incorrect:

```
String s = (String) 4.5;                  // this is wrong!
String t = "Value = " + (String) 13; // this is wrong!
String u = 22;                            // this is wrong!
```

To perform a conversion to a string, we must instead use the appropriate toString method or perform an implicit cast via the concatenation operation.

Thus, the following statements are correct:

```
String s = "" + 4.5;                  // correct, but poor style
String t = "Value = " + 13;           // this is good
String u = Integer.toString(22);      // this is good
```

A.4 Generics

A ***generic type*** is a type that is not defined at compilation time, but becomes fully specified at run time.

The generics framework allows us define a class in terms of a set of ***formal type parameters***, with could be used, for example, to abstract the types of some internal variables of the class. Angle brackets are used to enclose the list of formal type parameters. Although any valid identifier can be used for a formal type parameter, single-letter uppercase names are conventionally used. Given a class that has been defined with such parametrized types, we instantiate an object of this class by using ***actual type parameters*** to indicate the concrete types to be used.

In Code Fragment A.2, we show a class Pair storing key-value pairs, where the types of the key and value are specified by parameters K and V, respectively. The main method creates two instances of this class, one for a String-Integer pair (e.g., to store a dimension and its value), and the other for a Student-Double pair (e.g., to store the grade given to a student).

```
public class Pair<K, V> {
  K key;
  V value;
  public void set(K k, V v) {
    key = k;
    value = v;
  }
  public K getKey() { return key; }
  public V getValue() { return value; }
  public String toString() {
    return "[" + getKey() + ", " + getValue() + "]";
  }
  public static void main (String[] args) {
    Pair<String,Integer> pair1 = new  Pair<String,Integer>();
    pair1.set(new String("height"), new Integer(36));
    System.out.println(pair1);
    Pair<Student,Double> pair2 = new  Pair<Student,Double>();
    pair2.set(new Student("A5976","Sue",19), new Double(9.5));
    System.out.println(pair2);
  }
}
```

Code Fragment A.2: Example of use of generics.

The output of the execution of this method is shown below:

```
[height, 36]
[Student(ID: A5976, Name: Sue, Age: 19), 9.5]
```

In the previous example, the actual type parameter can be an arbitrary type. To restrict the type of an actual parameter, we can use an **extends** clause, as shown below, where class PersonPairDirectoryGeneric is defined in terms of a generic type parameter P, partially specified by stating that it extends class Person.

```
public class PersonPairDirectoryGeneric<P extends Person> {
  // ... instance variables would go here ...
  public PersonPairDirectoryGeneric() { /* default constructor goes here */ }
  public void insert (P person, P other) { /* insert code goes here */ }
  public P findOther (P person) { return null; } // stub for find
  public void remove (P person, P other) { /* remove code goes here */ }
}
```

Given the above class, we can declare a variable referring to an instance of PersonPairDirectoryGeneric that stores pairs of objects of type Student:

```
PersonPairDirectoryGeneric<Student> myStudentDirectory;
```

For such an instance, method findOther returns a value of type Student. Thus, the following statement, which does not use a cast, is correct:

```
Student cute_one =  myStudentDirectory.findOther(smart_one);
```

The generics framework allows us to define generic versions of methods. In this case, we can include the generic definition among the method modifiers. For example, we show below the definition of a method that can compare the keys from any two Pair objects, provided that their keys implement the Comparable interface:

```
public static <K extends Comparable,V,L,W> int
      comparePairs(Pair<K,V> p, Pair<L,W> q) {
  return p.getKey().compareTo(q.getKey());  // p's key implements compareTo
}
```

There is an important caveat related to generic types. Java allows for an array to be defined with a parameterized type, but it doesn't allow a parameterized type to be used to create a new array. Fortunately, it allows for an array defined with a parameterized type to then be initialized with a newly-created non-parametric array. Even so, this latter mechanism causes the Java compiler to issue a warning, because it is not 100% type-safe. We illustrate this point in the following:

```
public static void main(String[] args) {
  Pair<String,Integer>[] a = new Pair[10]; // right, but gives a warning
  Pair<String,Integer>[] b = new Pair<String,Integer>[10]; // wrong!!
  a[0] = new Pair<String,Integer>();  // this is completely right
  a[0].set("Dog",10);          // this and the next statement are right too
  System.out.println("First pair is "+a[0].getKey()+", "+a[0].getValue());
}
```

A.5 Iterators

A typical computation on a array list, list, or sequence is to march through its elements in order, one at a time, for example, to look for a specific element.

A.5.1 The Iterator and Iterable Abstract Data Types

An ***iterator*** is a software design pattern that abstracts the process of scanning through a collection of elements one element at a time. An iterator consists of a sequence S, a current element in S, and a way of stepping to the next element in S and making it the current element. Thus, an iterator extends the concept of the position ADT. In fact, a position can be thought of as an iterator that doesn't go anywhere. An iterator encapsulates the concepts of "place" and "next" in a collection of objects.

We define the ***iterator*** ADT as supporting the following two methods:

hasNext(): Test whether there are elements left in the iterator.

next(): Return the next element in the iterator.

Note that the iterator ADT has the notion of the "current" element in a traversal of a sequence. The first element in an iterator is returned by the first call to the method next, assuming of course that the iterator contains at least one element.

An iterator provides a unified scheme to access all the elements of a collection of objects in a way that is independent from the specific organization of the collection. An iterator for a array list, list, or sequence should return the elements according to their linear ordering.

Simple Iterators in Java

Java provides an iterator through its java.util.Iterator interface. We note that the java.util.Scanner class implements this interface. Interface java.util.Iterator supports an additional (optional) method to remove the previously returned element from the collection. This functionality (removing elements through an iterator) is somewhat controversial from an object-oriented viewpoint, however, and it is not surprising that its implementation by classes is optional. Incidentally, Java also provides the java.util.Enumeration interface, which is historically older than the iterator interface and uses names hasMoreElements() and nextElement().

The Iterable Abstract Data Type

In order to provide a unified generic mechanism for scanning through a data structure, ADTs storing collections of objects should support the following method:

iterator(): Return an iterator of the elements in the collection.

This method is supported, for example, by class java.util.ArrayList. In fact, method iterator() is so important, that there is a whole interface, java.lang.Iterable, which has only this method in it. This method can make it simple for us to specify computations that need to loop through the elements of a list. For example, we can define an alternative interface, called PositionList, for the list ADT, as shown in Code Fragment A.3. Interface PositionList includes is declared to extend interface Iterable.

```
public interface PositionList<E> extends Iterable<E> {
 // ...all the other methods of the list ADT ...
  /** Returns an iterator of all the elements in the list. */
  public Iterator<E> iterator();
}
```

Code Fragment A.3: Method iterator in the PositionList interface.

Given such a PositionList definition, we could use an iterator returned by the iterator() method to create a string representation of a node list, as shown in Code Fragment A.4.

```
  /** Returns a textual representation of a given node list */
  public static <E> String toString(PositionList<E> l) {
    Iterator<E> it = l.iterator();
    String s = "[";
    while (it.hasNext()) {
      s += it.next();   // implicit cast of the next element to String
      if (it.hasNext())
        s += ", ";
      }
    s += "]";
    return s;
  }
```

Code Fragment A.4: Example of a Java iterator used to convert a node list to a string.

In general, it is convenient to provide method iterator() in all interfaces representing collections of elements.

A.5.2 The Java For-Each Loop

Since looping through the elements returned by an iterator is such a common construct, Java provides a shorthand notation for such loops, called the ***for-each loop***. The syntax for such a loop is as follows:

for (*Type name* : *expression*)
    loop_statement

where *expression* evaluates to a collection that implements the java.lang.Iterable interface, *Type* is the type of object returned by the iterator for this class, and *name* is the name of a variable that will take on the values of elements from this iterator in the *loop_statement*. This notation is really just shorthand for the following:

for (Iterator<*Type*> it = *expression*.iterator(); it.hasNext();) {
    Type name = it.next();
    loop_statement
}

For example, if we had a list, values, of Integer objects, and the class of list values implements interface java.lang.Iterable, then we can add up all the integers in values as follows:

```
List<Integer> values;
// ... statements that create a new values list and fill it with Integers...
int sum = 0;
for (Integer i : values)
  sum += i;  // unboxing allows this
```

We would read the above loop as follows:

> "for each Integer i in values, do the loop body (in this case, add i to sum)."

In addition to the above form of for-each loop, Java also allows a for-each loop to be defined for the case when *expression* is an array (whose elements can have either a base type or an object type). For example, we can total up the integers in an array, v, which stores the first ten positive integers, as follows:

```
int[] v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total = 0;
for (int i : v)
  total += i;
```