CHAPTER 8 Collections and Generics

LEARNING OUTCOMES

By the end of this chapter, you should be able to:

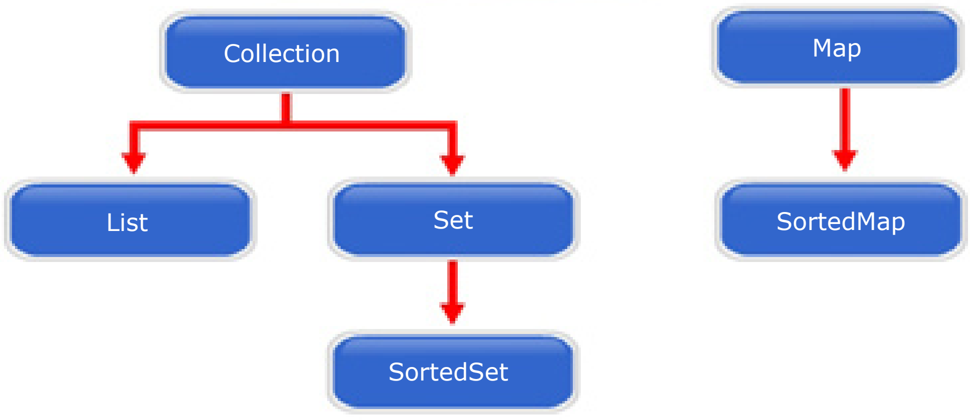
1. Define what text-wrapper class for primitive data type is;
2. Identify what collections in Java are;
3. Apply the Java collection framework to manipulate collections (such as search, sort and fill);
4. Use iterators to "walk through" a collection;
5. Recognise the concepts of generics in Java;
6. Create generic methods;
7. Identify how to overload generic methods; and
8. Use wildcard in generic methods.

INTRODUCTION

Data structures are one of the most crucial components in the design of software system. They are the essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large databases and directory systems. Traditional implementation of data structures in computer program is a painstaking task, in the sense that the process is often very "low level", involving complex manipulation of variable references and memory allocations. Therefore, it is not surprising that this task is one of the most difficult and error-prone activity.

Fortunately, the Java development environment supplies extensive prepackaged libraries of interfaces and algorithms for manipulating data structures. The libraries are categorised under the Java collections framework. The word collection is used as a common reference to data structures that hold objects that need to be operated upon together in some controlled fashion. Therefore, array is not considered as collection (although arrays play a role in the implementation of collections). In this chapter, we discuss the implementation and manipulation of data structure using the Java collections framework.

Java Collections Framework



Interfaces and Implementations of Collections and Map Interface

Interface	Implementation				Histrocal
Set	HashSet		TreeSet	Set	
List		ArrayList		LinkedList	Vector Stack
Map	HashMap		TreeMap		hashtable Properties

The discussion of collections will not be complete unless we also cover generics. Generics are salient feature of Java programming language where they allow programmers to create a "generic" type or method that operates on objects of various types while providing compile-time type safety. With generics, we can create a single sort method that could sort the elements in an integer array, a string array or an array of any type that support ordering. Generics complement the limitation of collections in which the lack of information about a collection's element type may result in the need to keep track of the type of elements collections contain, and the need for type casting.

8.1 TYPE-WRAPPER CLASSES FOR PRIMITIVE TYPES

Before we proceed to discuss collections and generics, it is important to understand the text-wrapper class. In Chapter 2, we have discussed primitive data types.

Primitive data types are special data types built into the language. They are not objects created from a class. Their name is part of the reserved keywords.

Each primitive type has a corresponding **type-wrapper class** (found in package `java.lang`). These are classes that enable you to manipulate primitive type values as objects. Many of the data structures and Java library classes require data in the form of object, therefore cannot manipulate primitive data type directly. The type-wrapper classes overcome this problem by "wrapping" the primitive types as object.

Figure 8.1 shows how each of the numeric type-wrapper classes overcomes the problem.

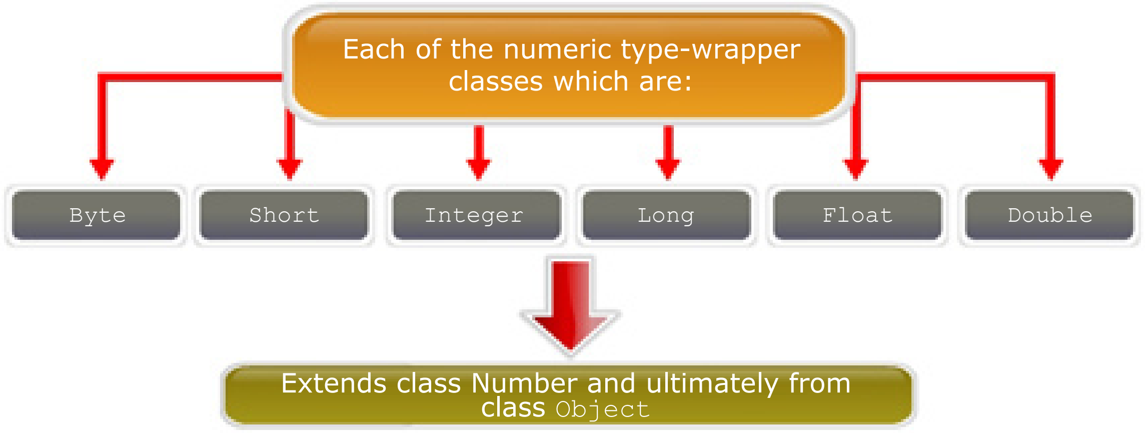

Each of the numeric type-wrapper classes which are:
Byte
Short
Integer
Long
Float
Double
Extends class Number and ultimately from class Object

Figure 8.1: Solution in how each of the numeric type-wrapper classes overcomes the problem

This enables you to take advantage of polymorphic behaviour of the type-wrapper classes. In addition, primitive types do not have method. On the contrary, the type-wrapper classes come with a handful of methods that allow you to manipulate the value (e.g. method `parseInt` converts a `String` to an `int` value, is located in type-wrapper class `Integer`).

8.2 COLLECTIONS

Collection is a special term used to refer to data structures that hold objects that need to be operated upon together in some controlled fashion.

The objects are usually of similar type, but the collection framework interfaces permits operations to be performed generically on any object derived from the superclass Object. Several implementations of basic collections interfaces are provided by the Java collections framework, those are shown in Table 8.1.

Table 8.1: Some Collection Framework Interfaces

Interface	Description
Collection	The root interface in the collections hierarchy from which all other collection interfaces are derived from.
List	A dynamic ordered list collection that can contain duplicate elements.
Set	A dynamic list that does not allow duplicates.
Map	A hash table implementation of collection.
Queue	Typical first-in, first out collection.
SortedSet	A Set that maintains its elements in ascending order.
SortedMap	A Map that maintains its mappings in ascending key order.

Table 8.1 shows each collection item has rules that govern how the elements in the collection should be operated. Therefore an array is not considered as collection because there are simply no rules regarding how the elements of an array should be organised or behave.

8.2.1 Benefits of the Java Collections Framework

Using the Java Collections Framework in application development poses several benefits as listed in Figure 8.2.

- It reduces programming effort. The readymade data structures and algorithms in the Collections Framework free programmers from the drudgery task of implementing the data structures, thus allowing programmers to devote their effort on the important parts of the program.
- Using the Collections Framework increases program speed and quality, and reduces bugs in programs. The algorithms in the Collections Framework are well implemented by professional programmers and tested by thousands of users around the world. These ensure high-performance, high-quality and bugs free implementations of the algorithms.
- The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations.
- The Collections Framework facilitates interoperability among unrelated APIs, thus frees programmers from writing adapter objects or conversion code to connect APIs. This ensures seamless interoperability, even though the APIs were written independently by different programmers.
- It reduces effort to learn and to use new APIs as the Collections Framework provides a consistent way on how each API works with collection.
- Last but not least, the Collections Framework fosters software reuse.

Figure 8.2: Several benefits of the Java Collections Framework

8.2.2 Interface Collection

Interface `Collection` is the root interface in the collection hierarchy from which the interface Set, Queue and List are derived from.

It contains the common methods that are shared by all the collections objects such as shown in Figure 8.3.

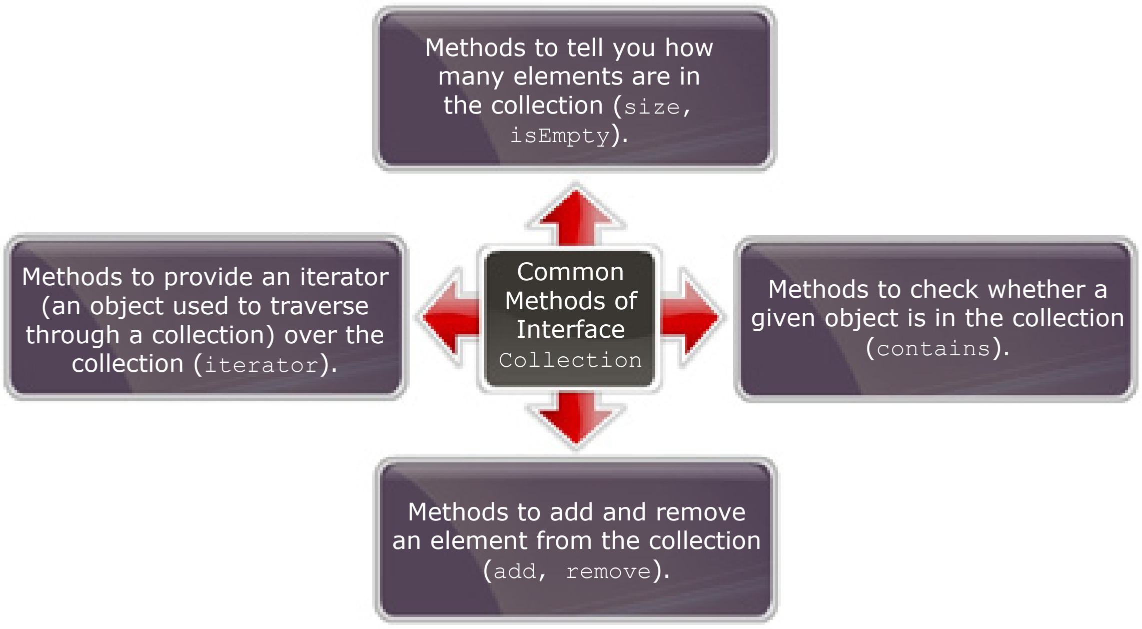

Methods to tell you how many elements are in the collection (size, isEmpty).
Methods to provide an iterator (an object used to traverse through a collection) over the collection (iterator).
Common Methods of Interface Collection
Methods to check whether a given object is in the collection (contains).
Methods to add and remove an element from the collection (add, remove).

Figure 8.3: Common methods of interface Collection

Interface collection also contains **bulk operations** (i.e. operations performed on an entire collection) for adding, removing, clearing, comparing and retaining objects in a collection. Although you could provide your own implementation of these operations, in most cases such implementations would be less efficient as compared to the built-in bulk operations.

Finally, the interface Collection provides methods to convert a collection into an array (`toArray`). This is a bridge between collections and older APIs that expect input as array.

8.3 LIST

List and dynamic can be defined as shown in Figure 8.4.

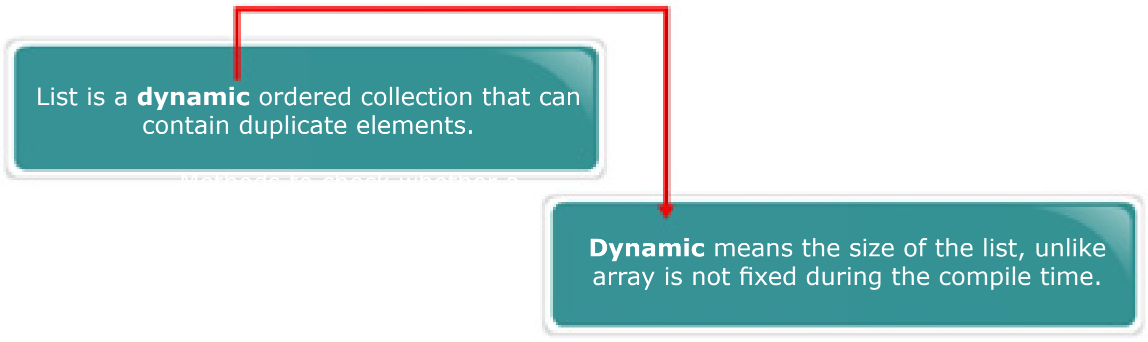

List is a **dynamic** ordered collection that can contain duplicate elements.
Dynamic means the size of the list, unlike array is not fixed during the compile time.

Figure 8.4: Definition of list and dynamic

Nonetheless, elements in the list are indexed with the first index starts from zero. The `List` interface extends the functionalities of `Collection` by providing methods for manipulating elements via their indices, manipulating a specified range of elements, searching for elements and getting a `ListIterator` to access the elements.

In practice, we rarely implement `List` in our application (unless if we are developing our own custom `List` collection class); rather, we use the three general-purpose concrete `List` implementations provided by the Java platform, i.e. the `ArrayList`, the `LinkedList` and the much earlier `Vector` class which has been retrofitted to implement `List`. Which one you should choose will depend on the requirements of your application. The primary difference between these implementations is at the low-level design aspects of the list and not in term of their applications. Figure 8.5 lists the differences between three general-purpose concrete List implementations.

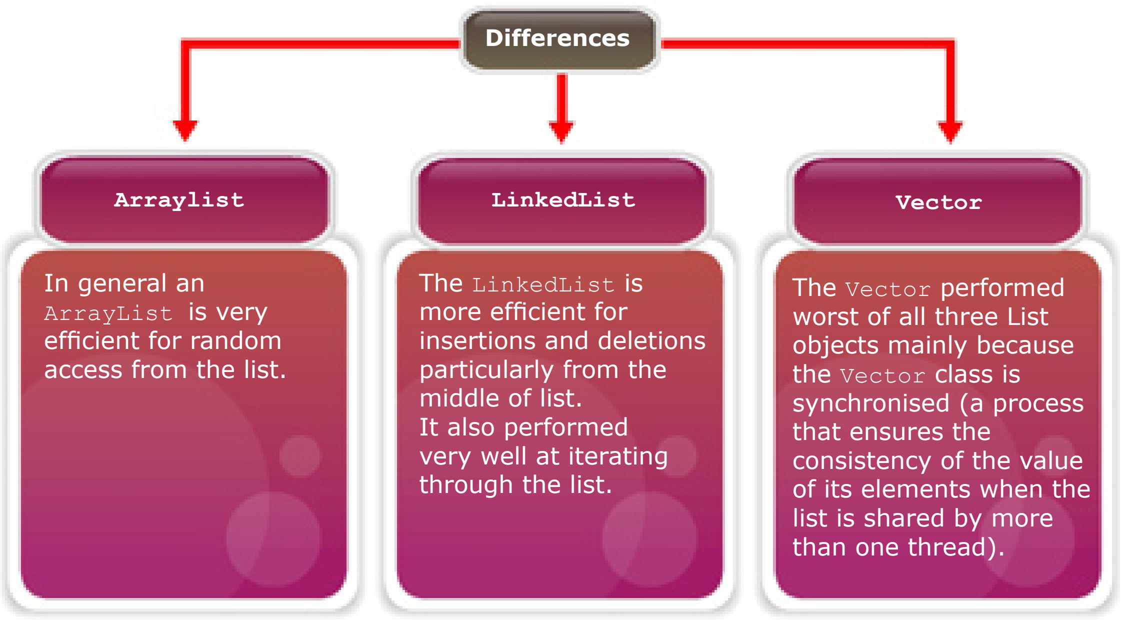

Differences
Arraylist
LinkedList
Vector
In general an ArrayList is very efficient for random access from the list.
The LinkedList is more efficient for insertions and deletions particularly from the middle of list. It also performed very well at iterating through the list.
The Vector performed worst of all three List objects mainly because the Vector class is synchronised (a process that ensures the consistency of the value of its elements when the list is shared by more than one thread).

Figure 8.5: Differences between three general-purpose concrete List implementations

8.3.1 ArrayList

Using the Java Collections Framework in application development poses several benefits as listed in Figure 8.2.

`ArrayList` is the resizable-array implementation of the `List` interface.

It is an ideal data structure if you have no knowledge in advance about elements number, but performance wise is slower than array. `ArrayLists` internally **implemented as an Array**. So when a new element is added, first, the system will create a new array with *n*+1 elements. All existing elements will be copied to first *n* elements and last *n*+1 will be filled with the new element. To avoid that internal re-copying of Arrays you can use `ensureCapacity (int requestCapacity)` method to create an array with required capacity only once therefore improving its performance.

Program in Figure 8.6 demonstrates a simple `ArrayList` application. The application creates a list based on user input (in the program execution arguments) and calls the method shuffle from the Java's platform Collections class to randomly permute the element of the list and then writes out the content of the list.

```
//Shuffle.java
//Program to demonstrate List
import java.util.*;

public class Shuffle
{
 public static void main(String[] args)
 {
    List<String> list = new ArrayList<String>();
        for (String a : args)
             list.add(a);
    Collections.shuffle(list, new Random());
    System.out.println(list);
 }
}
```

Output:

```
>java Shuffle 1 2 3
       4 5
 [5, 2, 1, 3, 4]
```

Figure 8.6: Simple list application

There are several important aspects of list as well as Java application that are shown in the program in Figure 8.6. The declaration of `List` uses the `ArrayList` implementation and declares a list of type `String` as shown in Figure 8.7.

```
List<String> list = new ArrayList<String>();
```

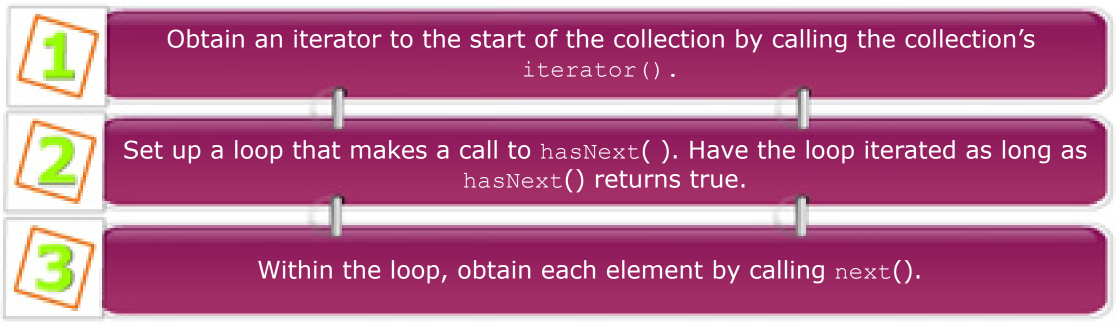
Figure 8.7: Implementation and declares a list of type `String`

The content of the list is the arguments provided with the execution of the program (store in the `String` array `args`). The application then calls the shuffle method from the Java's platform `Collections` class to randomly permute the element of the list and then write out the content of the list. What is interesting is that the `println` method automatically formats the list element in between a pair of square braces with comma separators between each element.

8.3.2 Using Iterator

`Iterator` is an object that allows you to walk through a collection.

The collection classes provide an `iterator()` method that returns an iterator to the start of the collection. In general, Figure 8.8 shows the steps involved in using an iterator to walk through the contents of a collection.



1. Obtain an iterator to the start of the collection by calling the collection's `iterator()`.
2. Set up a loop that makes a call to `hasNext( )`. Have the loop iterated as long as `hasNext()` returns true.
3. Within the loop, obtain each element by calling `next()`.

Figure 8.8: Steps involved in using an `iterator`

The Java code using Iterator is shown in Figure 8.9.

```
Iterator itr = some_collection.iterator();
while(itr.hasNext())
{
    Object element = itr.next();
    //do stuff with element
}
```

Figure 8.9: The Java code using `Iterator`

In List, you can obtain a ListIterator by calling the `listIterator()` method. A list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, `ListIterator` is used just like an `Iterator`.

The question that probably comes to our mind is why using an `Iterator` when we can just walk through a `Collection` using a for loop just like the codes shown in Figure 8.10.

```
     for (int i = 0; i < some_collection.size(); i++)
     {
         Object x = some_collection[i];
     }
```

Figure 8.10: The codes using a `for` *loop*

Table 8.2 shows the reasons to why an interator is used.

Table 8.2: Reasons to Why an `Iterator` *is used*

Answer	Explanation
Efficiency	The above codes have the assumption that the method `size()` is a fast operation, which may not be the case for some collections (such as `List`). The same for assumption that accessing an element in a collection using the [] index is a fast operation, which again may not be the case for some collections (such as `List`). Using an iterator brings you closer to collection independence, without making assumptions on the efficiency of random access ability of the collection as well as that of the `size()` operation.
Reduce potential error	An iterator basically removes any potential risk of stack overflow, (i.e. accessing element outside the bound of collection). Such may not be the case using a `for` loop. In a large collection, `int` may not be large enough for your collection's indices which may result in unforeseeable error at runtime.
Separation of Concerns	Using iterator separates the iteration code from the 'core' concern of the loop.
Declarative Implementation	Using iterator unlocked the declarative power of Java code where programmers only concern with what needs to be done and not how it should be done.

In the next example, we demonstrate an application that manipulates list of data using ArrayList. It is a simple application that allows user to select items from one list into another. You can download, compile and run the code for this application, *ArrayListDemo.java.* Figure 8.11 shows the snapshot of the application.

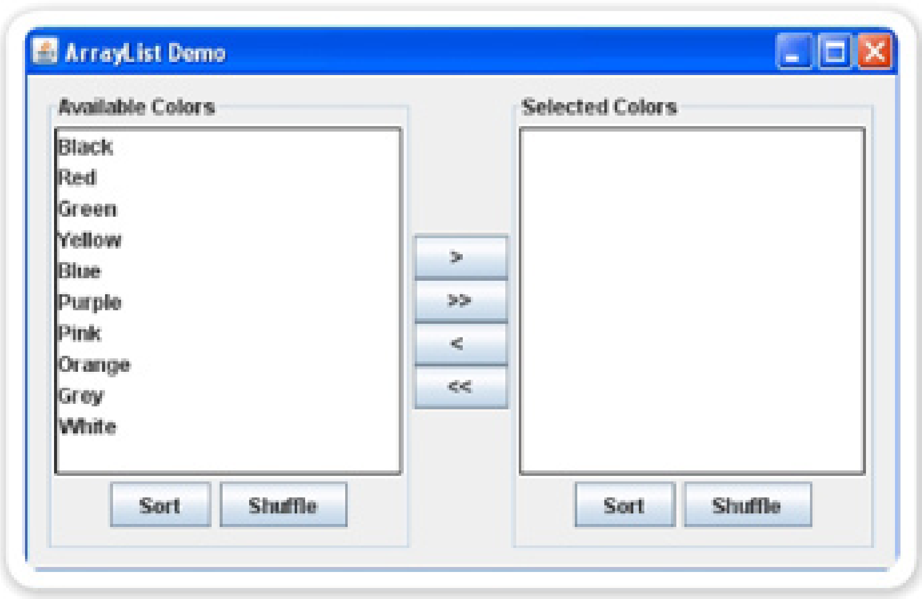


Figure 8.11: Arraylist Demo - color chooser application

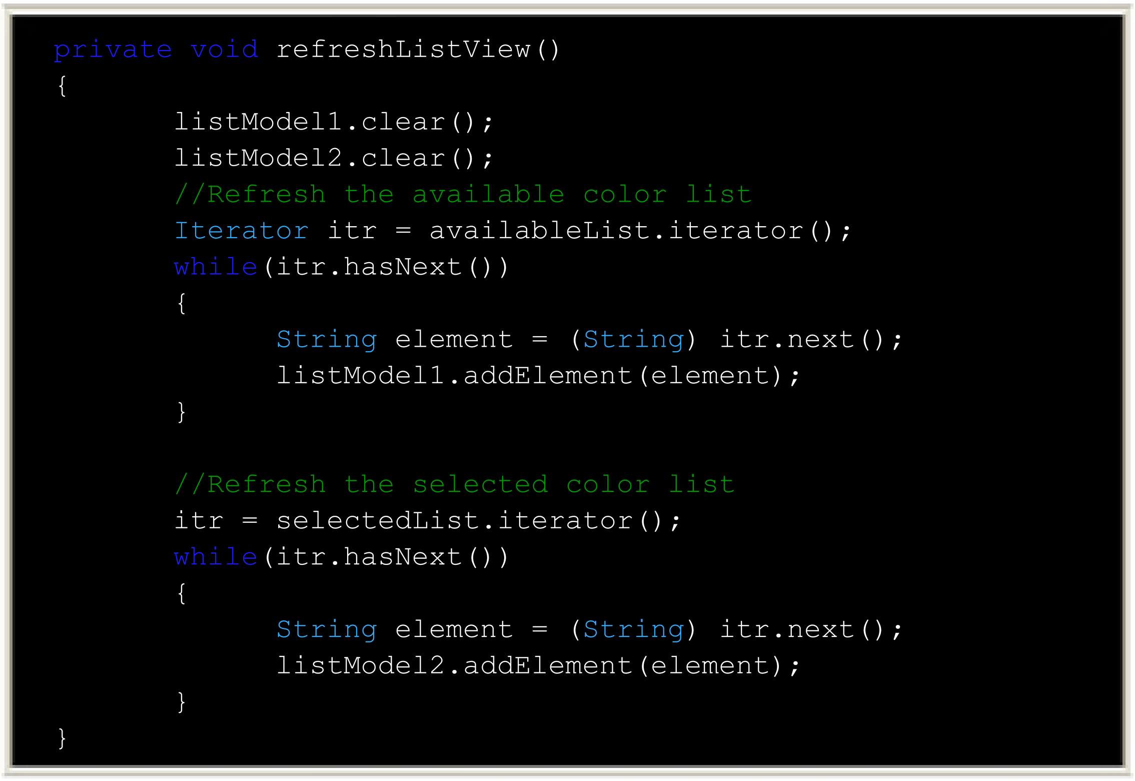
The application in Figure 8.11 demonstrates a typical item chooser in a GUI application. User selects items from the list box on their left and the selected item will appear on the list box on their right. In addition to selecting items, the application allows user to sort or shuffle the items in the list boxes.

The application uses two `ArrayList` to store the data as shown in Figure 8.12.

```
List<String> availableList = new ArrayList<String>();
List<String> selectedList = new ArrayList<String>();
```

Figure 8.12: The code of two `ArrayList` to store the data

The `availableList` stores the list of available colors and the `selectedList` stores the colors selected by the user. Method `refreshListView()` fills the list boxes with the values of the `ArrayList`. It uses an iterator to walk through each list. Figure 8.13 shows the code of `refreshListView()` method.



```
private void refreshListView()
{
        listModel1.clear();
        listModel2.clear();
        //Refresh the available color list
        Iterator itr = availableList.iterator();
        while(itr.hasNext())
        {
              String element = (String) itr.next();
              listModel1.addElement(element);
        }

        //Refresh the selected color list
        itr = selectedList.iterator();
        while(itr.hasNext())
        {
              String element = (String) itr.next();
              listModel2.addElement(element);
        }
}
```

Figure 8.13: The code of `refreshListView()` method

The application also demonstrates the use of common list methods, such as `add` (to add an element to the list), `remove` (to remove an element from the list), `clear` (bulk operation that empties the list), `addAll` (bulk operation that copies from one list to another), `Collections.sort` (to sort the list in ascending order) and `Collection.shuffle` (to randomly shuffle the elements in the list).

NOTES

`Collections` is a class consists exclusively of static methods with polymorphic algorithms that operate on collections, “wrappers”, which return a new collection backed by specified operations.

8.3.3 LinkedList

We have explained earlier that `ArrayList` internally is implemented as an array. Array is a good data structure for random access of its element, but performs poorly in insertion and deletion of element. `LinkedList` on the other hand does not use arrays. It is implemented using a doubly-linked list where each node contains two links or pointers to next or previous node. A `LinkedList` looks like a chain, consisting of people who hold each other's hand. It permits node insertion and removal operations at any point in the list in constant time. Neverthe-less, accessing an element requires a sequential scan from the front or back of the list to the location of the element in the list. There is no fast way to access the *N*th element of the list as elements in the list are not stored in contiguous plot of memory accessible by computing the offset of its indices as found in array structures.

Which structure is a better choice?
There is no definite answer. It depends on the requirements of your program. There are many cases where LinkedList does better and vice versa

You can download, compile and run the code for this application, *LinkedListDemo.java*. Table 8.3 shows the results of comparison of performance between `ArrayList` and `LinkedList` on the `add`, get and binary search operations.

Table 8.3: Comparison of Performance between `ArrayList` *and* `LinkedList`

```
------------------------------------------------------------------------
        Operation              -    ArrayList      LinkedList
------------------------------------------------------------------------
    add Element 0th            -     218 ms            0 ms
                               -
    get Element 5000th         -       0 ms          203 ms
                               -
    search Element             -      16 ms         1563 ms
------------------------------------------------------------------------
```

The application creates the respective lists with 10000 elements. The add operation simply adds a new element at the beginning of the list (0th element) and the get operation obtain the element of the 5000th. The binary search operation simply searches for the existence of an element in the list using the `BinarySearch` method in the `Collections` class. All operations are repeated for 10000 times to magnify the execution time, which is measured in millisecond.

The results in Table 8.3 are predictable. `ArrayList` takes longer time to insert element due to its "array" nature. When an element is added to the beginning of an `ArrayList`, all of the existing elements must be pushed back, which means a lot of expensive data movement and copying. Conversely, LinkList can add an element to the beginning of the list efficiently, simply by means of allocating an internal record for the element and then adjusting a couple of links.

The results in obtaining the 5000th element are pretty much the reverse. LinkedList performs poorly in contrast to the ArrayList. The reason is because for each retrieval, the LinkedList get algorithm has to traverse the list from the 0th element to the 5000th whereas an ArrayList can simply retrieve the element by the offset of the location of the 5000th element from the 0th element. This is also the reason that explains why ArrayList performs significantly better than LinkedList when it comes to binary search. The binary search algorithm inherently uses random access, and LinkedList does not support fast random access.

8.3.4 Vector

The Vector class was included in Java prior to the development of the Collection and List interfaces. It has been retrofitted into the List interface without removing the methods which were originally in the class. For instance Vector has methods add and addElement in which both perform the same function – appending an element to Vector. But only add method is specified in interface List. When you use the Vector you should use only the methods found in the List interface so that it will be easy to switch to another List object if it becomes desirable.

The implementation and application of Vector is pretty much similar to ArrayList (i.e. a growable array of objects) except that the Vector class is synchronised. Synchronisation is a process to ensure the consistency of the value of its elements when the list is shared by more than one thread.

One of the well-known structures that implement the Vector collection is the built-in Stack data structure under the Stack class (java.util.Stack). A stack is a last-in-first-out (LIFO) data structure. The built-in Stack class extends class Vector with five basic stack operations as shown in Figure 8.14 that allow a vector to be treated as a stack.

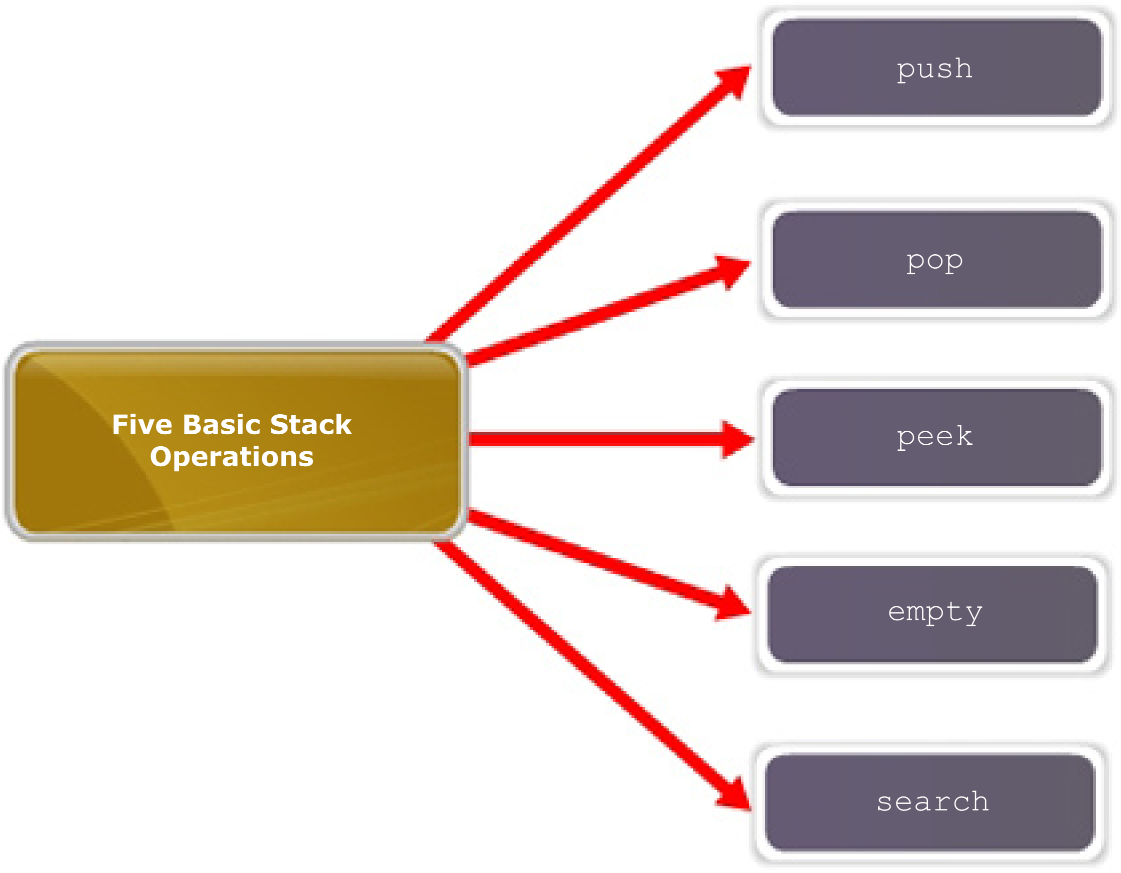


Figure 8.14: Five basic stack operations

8.4 MAP

Map is a data structure where keys are mapped to specific value.

Each key can map to at most one value. Conceptually, you could consider Lists as Maps which have numerical keys. Numerical keys are only ideal when you want to access elements by their numerical index with the condition that the element's data matches a numerical key value and the keys are unique and tightly packed. However if you have 100 employees with five-digit employee ID and you would like to store and retrieve data using the employee ID as key, the task would require an array of 10000 elements because there are 10000 unique five-digit numbers. This is totally impractical. A map provides the solution to this problem by allowing a generic key to be transformed into an array index via a function of mathematical abstraction. The transformed key can then be mapped to their respective value efficiently as shown in Figure 8.15.

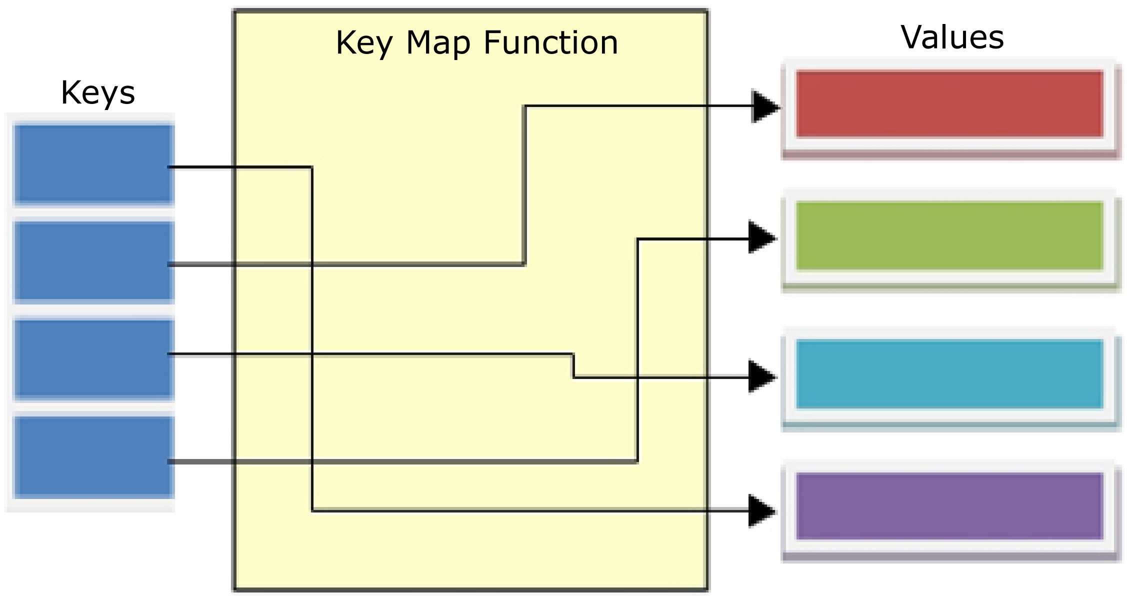


Figure 8.15: Map visualisation

The Java platform provides three general concrete implementations of Map interface which are shown in Figure 8.16.

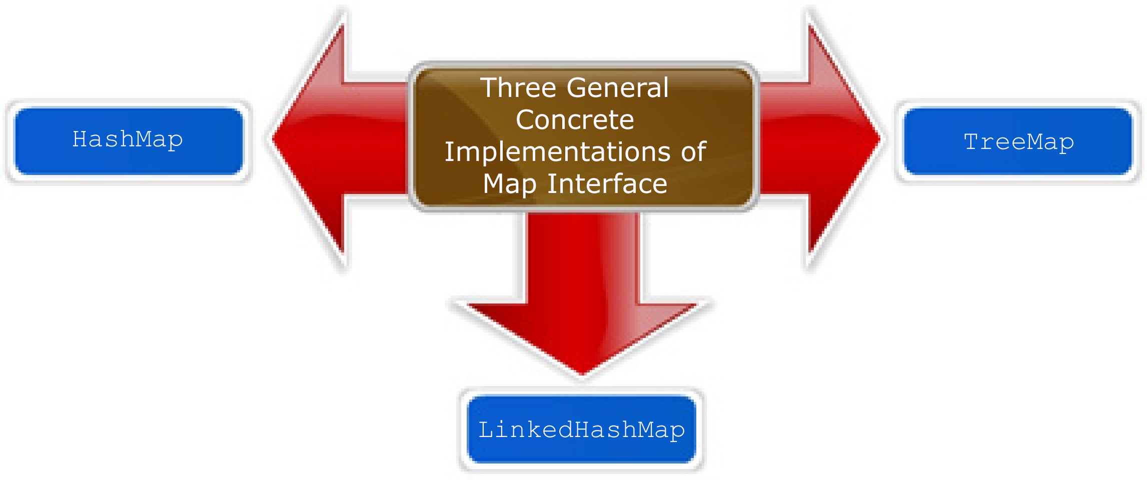
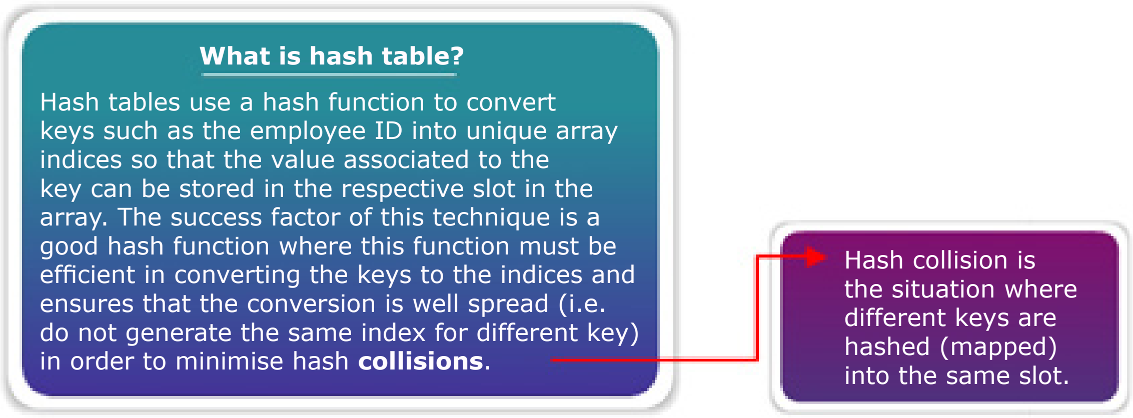


Figure 8.16: Three general concrete implementations of Map interface

Also, the earlier `Hashtable` was retrofitted to implement Map. As their names imply, `HashMap`, `LinkHashMap` and `Hashtable` store elements in hash tables and `TreeMaps` store elements in trees. As you may be aware by now that the many flavours of Map concrete implementations differ in their internal working mechanism and not on their superficial appearance to the programmers. In this section, we will discuss hash tables and provides an example that uses a `HashMap`.

Figure 8.17 shows the definition of hash table.



What is hash table?

Hash tables use a hash function to convert keys such as the employee ID into unique array indices so that the value associated to the key can be stored in the respective slot in the array. The success factor of this technique is a good hash function where this function must be efficient in converting the keys to the indices and ensures that the conversion is well spread (i.e. do not generate the same index for different key) in order to minimise hash **collisions**.

Hash collision is the situation where different keys are hashed (mapped) into the same slot.

Figure 8.17: Definition of hash table

Collision has to be resolved using collision resolution technique which in turn will increase the overhead of the algorithm. The detailed implementation of hash table is beyond the scope of this course. Computer science students study hash table in the subject algorithms and data structures. The reality is that hash tables are complex program. The `Hashmap`, `LinkHashMap` and `Hashtable` class enable programmers to use hashing without having to implement hash table mechanisms. This is the profound advantage of object oriented programming where classes encapsulate and hide complexity (i.e. implementation details) and offer user-friendly interfaces.

You can download, compile and run the code for this application, *HashMapDemo.java.* The application uses a `HashMap` to store the words. The method that creates the `HashMap` is the `createMap` method.

```java
private void createMap(String input)
{
        map = new HashMap<String, Integer>();
        StringTokenizer tokenizer = new StringTokenizer(input);
        int count;
        String word;
        //Processing the text
        while(tokenizer.hasMoreTokens())
        {
              word = tokenizer.nextToken().toLowerCase();//get word
              if(map.containsKey(word)) //existing word
              {
                    count = map.get(word); //get current word count
                    map.put(word, count + 1); //increase the count
              }
              else //new occurance
                   map.put(word, 1); //add new word
        }
}
```

Figure 8.18: The HashMap code

The strategy is to use each word as the key and the number of occurrence as the value of the key. The `createMap` method uses the string tokeniser object to obtain word by word from a string and call the method:

```
map.containsKey(word)
```

to find out if the word exists in the `HashMap`. Method `containsKey(word)` returns `true` if the word exists in the `HashMap` and `false` if it doesn't. If the word does not exist, we create a new entry of the map by using the word as the key and integer 1 as the value mapped to the key. The method `put(<key>, <value>)` creates a new entry of the map as shown in Figure 8.19.

```
map.put(word, 1);
```

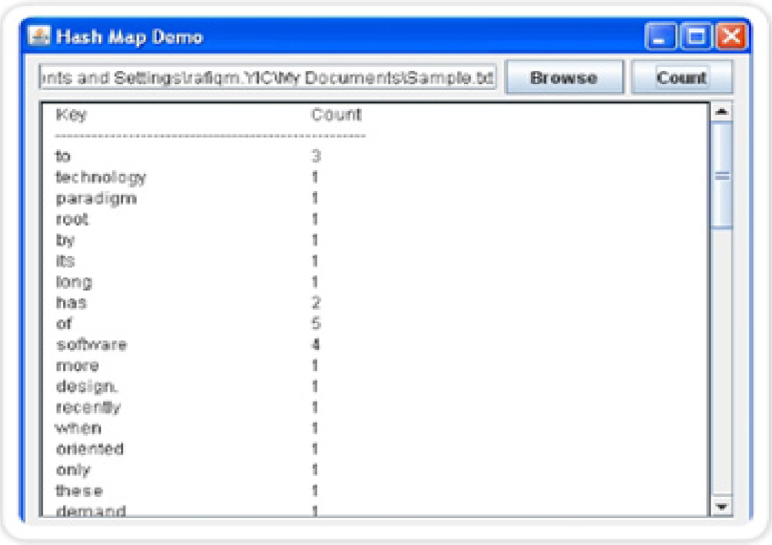
Figure 8.19: New entry of the map using method `put`

If the word exists, we obtain the value of the key using the function `get(<key>`. Then we put the key back after increasing the word count by one as shown in Figure 8.20.

```
count = map.get(word);
map.put(word, count + 1);
```

Figure 8.20: Value of the key using the function `get`

The example in Figure 8.21 demonstrates the use of `HashMap`. The application reads a text file and counts the number of occurrences of each word in the text file.



```
Key                             Count
------------------------------------------
to                              3
technology                      1
paradigm                        1
root                            1
by                              1
its                             1
long                            1
has                             2
of                              5
software                        4
more                            1
design.                         1
recently                        1
when                            1
oriented                        1
only                            1
these                           1
demand                          1
```

Figure 8.21: Hash Map Demo - word counting application

The method `displayMap` displays the result by presenting the content of the `HashMap` as shown in Figure 8.21.

```
private void displayMap()
{
        textArea.setText("Key\t\tCount\n"); //initialize the text area
        textArea.append("----------------------------------------\n");
        Set<String> keys = map.keySet(); //get the keys of the map
        for(String aKey: keys)
             textArea.append("" + aKey + "\t\t" + map.get(aKey) + "\n");
}
```

Figure 8.21:Content of the `HashMap`

First we get all the keys of the `HashMap` using the method `keySet()` and store them in a `Set` as shown in Figure 8.22.

```
Set<String> keys = map.keySet();
```

Figure 8.22: Keys of the HashMap using the method `keySet()`

Then we iterate through the set and output the keys and their respective values in the `HashMap` using the method using method `get(<Key>)`.

The application we have presented does not only demonstrate the use of `HashMap`, but also include several important features of Java. This includes two new GUI components, the text area (`JTextArea`) and file chooser (`JFileChooser`). We have also used the `StringTokenizer` class to tokenise string into words and `FileInputStream` to read from a text file.

8.5 GENERICS

Up to this point, we have been using Generics in our examples on List and Maps. When we declare a list using the following declaration:

```
List <String>list = new ArrayList<String>();
```

We have actually declared a generic list that can operate on type `String`. All generic method declarations have a type parameter section delimited by angle brackets (< and >). The type parameter section contains one or more types separated by comma. The actual plain List declaration without generic is simply a typical object declaration:

```
List list = new ArrayList();
```

Why do we want to declare a list as generic when we can just declare it as an ordinary object?

The answers lie in the following factors:

(a) Compile-time type safety – detecting type mismatch at compile time.
Example: if a list stores only integers, attempting to add a string into the list should issue a compile time error. In a non-generic implementation (i.e. via inheritance and method overloading), it does not happen. For instance, a list can accept any types of objects because all classes are derived from superclass `Object`. If type mismatch is not detected at compile time, the effect may be devastating during runtime. Consider the code snippet as shown in Figure 8.23.

```
1. int i;
2. List list = new ArrayList();
3. for (i = 0; i < 10; i++)
4. list.add(new Integer(i));
5. list.add(new String("A"));
6. Collections.shuffle(list, new Random());
7. Iterator itr = list.iterator();
8. while(itr.hasNext())
9. {
10. Integer element = (Integer) itr.next();
11. System.out.print(element.toString() + " ");
12. }
```

Figure 8.23: The code snipet of compile-time type safety

The codes above do not generate any error during compile time although we inserted object of type `Integer` (in line 4) and String (in line 5) into the list. But when the program is executed, it will throw an exception (in line 10) when the object of type `String` is encountered.

(b) Generic can overcome complicated overloaded methods to perform similar operations on different types of data. Consider the program in Figure 8.24. The program prints the content of array. We use many overloaded methods in order to allow the `printArray` method to work seamlessly on any arbitrary data type. But such implementation is lengthy, repetitive and cumbersome because the same method has to be written repeatedly though the codes only differ in term of the data type.

```
//OverloadedDemo.java
//Program to demonstrate method overloading and generic
import java.util.*;

public class OverloadedDemo
{
       public static void printArray(Integer[] input)
       {
              for(Integer element : input)
                     System.out.printf("%s ", element);
                     System.out.println();
      }

      public static void printArray(Double[] input)
      {
              for(Double element : input)
                     System.out.printf("%s ", element);
              System.out.println();
      }

      public static void printArray(Character[] input)
      {
              for(Character element : input)
                     System.out.printf("%s ", element);
              System.out.println();
      }

      public static void main(String[] args)
      {
             Integer[] intArray = {1,2,3,4,5,6,7,8,9};
             Double[] dobArray = {1.1, 2.2, 3.3, 4.4, 5.5};
             Character[] chrArray = {'J', 'A', 'V', 'A'};
             System.out.println("Integer array:");
             printArray(intArray);
             System.out.println("Double array:");
             printArray(dobArray);
             System.out.println("Character array:");
             printArray(chrArray);
      }
}
```

Figure 8.24: `printArray` using method overloading

The same program can be rewritten using generic. The array and element type are replaced with a generic name (i.e. E). The `printArray` method would look like the one in Figure 8.25 where only one `printArray` method is required and it can accept any arbitrary data type.

```
//OverloadedDemo.java
//Program to demonstrate method overloading and generic
import java.util.*;

public class GenericOverloadedDemo
{
       //Generic method
       public static <E> void printArray(E[] input)
       {
              for(E element : input)
                    System.out.printf("%s ", element);
              System.out.println();
       }

       public static void main(String[] args)
       {
              Integer[] intArray = {1,2,3,4,5,6,7,8,9};
              Double[] dobArray = {1.1, 2.2, 3.3, 4.4, 5.5};
              Character[] chrArray = {'J', 'A', 'V', 'A'};
              System.out.println("Integer array:");
              printArray(intArray);
              System.out.println("Double array:");
              printArray(dobArray);
              System.out.println("Character array:");
              printArray(chrArray);
       }
}
```

Figure 8.25: printArray using method overloading

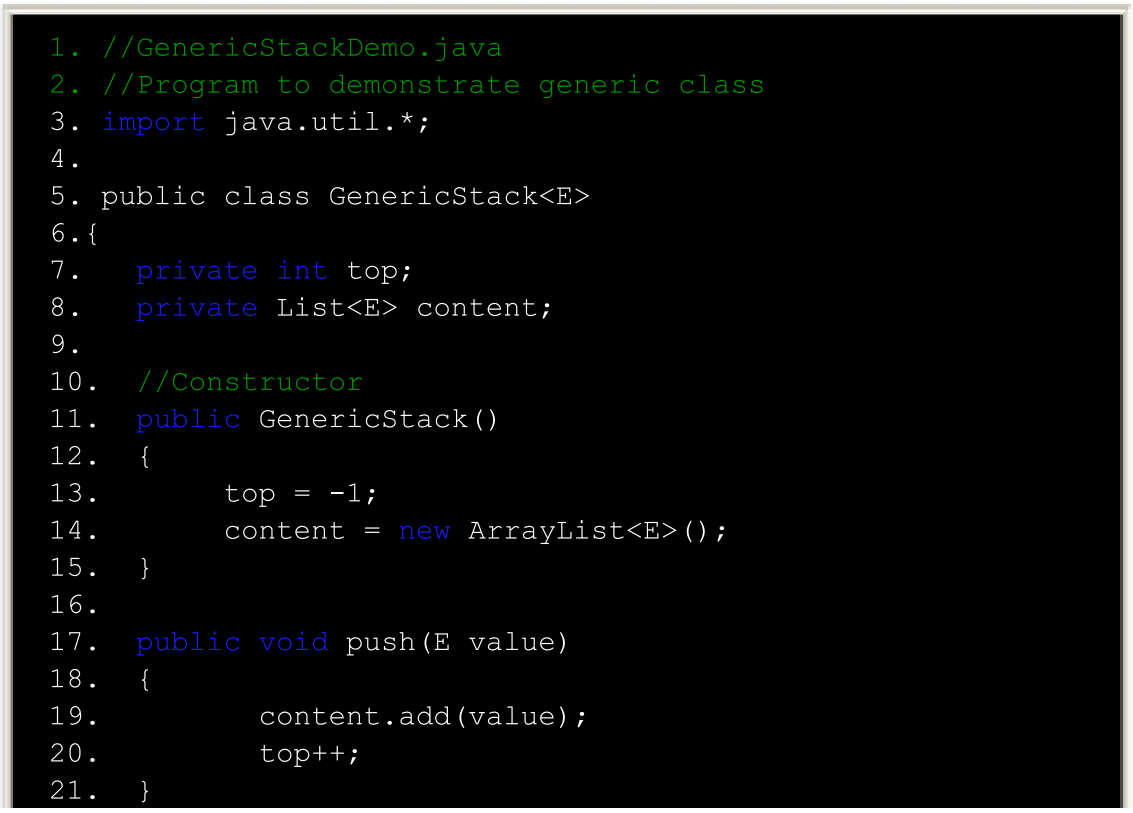
The motivation for using generics is to overcome the shortcoming of inheritance where thelack of information about an object type requires developers to keep track of the type and the need for casts all over the place. This is inconvenient and unsafe as the compiler does not check that your cast is the same as the object type, so the cast can fail at run time. Generics provide a way to communicate the type of object to the compiler, so that it can be checked. Once the compiler knows the object type, the compiler can check that you have used the object consistently.

Generic overcomes the need to overload methods that accept different data types. Retrospectively it invalidates the ability to overload methods by using parameter of different data types. What if you need to overload the generic methods? The only way to overload generic methods is by using different number of arguments. For instance, the method `printArray` can be overloaded with additional parameters such as `arraySize`. Also, a generic method will be overloaded by non-generic methods with the same name but different number of arguments.

8.5.1 Generic Classes

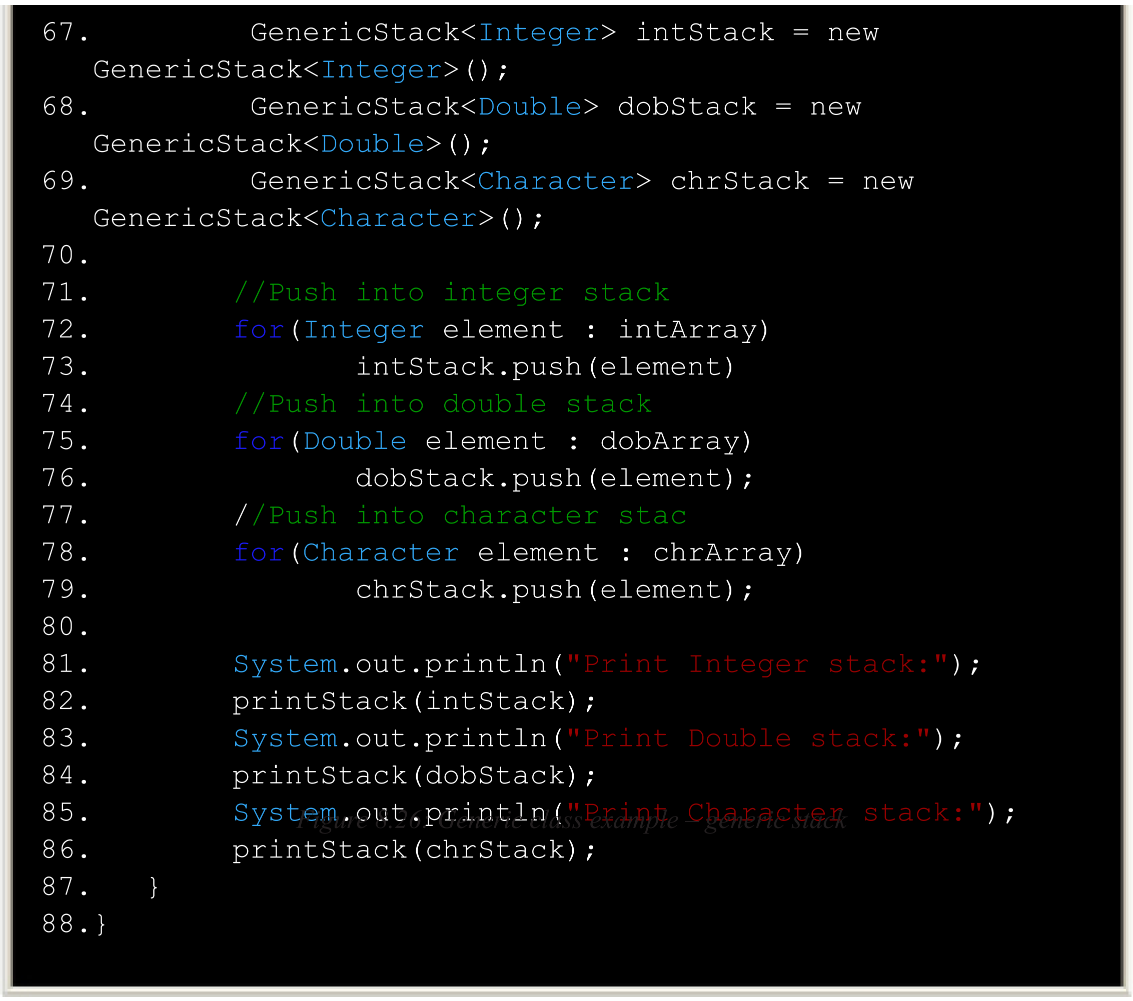
The concept of data structure such as list, stack, queue and tree can be understood independently of the element type they manipulate. For instance we can have a stack of `Double`, a stack of `Integer`, a stack of `String` and a stack of `Employee`. The codes for such classes differ only in the type of object they manipulate. One common way to write such classes is to exploit the polymorphic behaviour of object by using an object which is the superclass of all the objects that can be manipulated by the data structure classes. This however is not type-safe and requires consistent type casting as we have discussed earlier.

A better way to implement such classes is to use the generic class. A generic class provides a means to implement some functionality as class in a type-independent manner. The class can later be instantiated with type-specific object of the generic class. At compilation time, the Java compiler ensures the type safety of your code. Figure 8.26 demonstrates the use of generic class in creating a generic stack class.



```
//GenericStackDemo.java
//Program to demonstrate generic class
import java.util.*;

public class GenericStack<E>
{
 private int top;
 private List<E> content;

 //Constructor
 public GenericStack()
 {
    top = -1;
    content = new ArrayList<E>();
 }

 public void push(E value)
 {
     content.add(value);
     top++;
 }
```

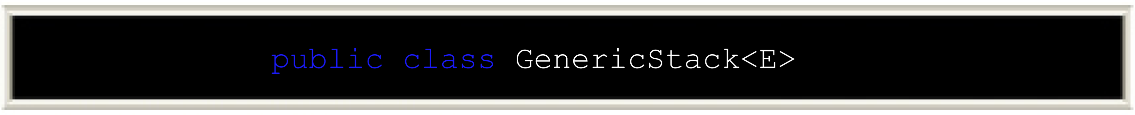
```java

 public E pop()
 {
        if(top == -1)
        {
               System.out.println("Stack empty");
               return null;
        }
        else
        {
               E retVal = content.get(top);
               content.remove(top);
               top--;
               return retVal;
        }
  }


  public boolean isEmpty()
  {
         if(top == -1)
              return true;
         else
              return false;
  }
}

  //GenericStackDemo.java
  //Program to demonstrate generic class
  import java.util.*;
  public class GenericStackDemo<E>
  {
       public static void printStack(GenericStack aStack)
       {
               while(!aStack.isEmpty())
                  System.out.printf("%s ", aStack.pop());
               System.out.println();
  }

  public static void main(String[] args)
  {
        Integer[] intArray = {1,2,3,4,5,6,7,8,9};
        Double[] dobArray = {1.1, 2.2, 3.3, 4.4, 5.5};
        Character[] chrArray = {'H', 'E', 'L', 'L', 'O'};

```



```java
        GenericStack<Integer> intStack = new
GenericStack<Integer>();
        GenericStack<Double> dobStack = new
GenericStack<Double>();
        GenericStack<Character> chrStack = new
GenericStack<Character>();

        //Push into integer stack
        for(Integer element : intArray)
                intStack.push(element)
        //Push into double stack
        for(Double element : dobArray)
                dobStack.push(element);
        //Push into character stac
        for(Character element : chrArray)
                chrStack.push(element);

        System.out.println("Print Integer stack:");
        printStack(intStack);
        System.out.println("Print Double stack:");
        printStack(dobStack);
        System.out.println("Print Character stack:");
        printStack(chrStack);
    }
}
```

Figure 8.26: Generic class example – generic stack

The generic class declaration has a pair of type parameter section delimited by angle brackets (< and >). `E` is just an identifier similar to method parameters, except we do not specify the type of `E` as shown in Figure 8.27.



```java
public class GenericStack<E>
```

Figure 8.27: E as identifier similar to method parameters

The generic class declaration has a pair of type parameter section delimited by angle brackets (< and >). `E` is just an identifier similar to method parameters, except we do not specify the type of `E` as shown in Figure 8.28.

```
GenericStack<Integer> intStack = new GenericStack<Integer>();
GenericStack<Double> dobStack = new GenericStack<Double>();
GenericStack<Character> chrStack = new GenericStack<Character>();
```

Figure 8.28: Three GenericStack objects

Note that the type of object for the generic class is given in the angle brackets. The compiler will assure that the object instantiated can only be applied to their respective type. Attempting to push a `Double` value to the `intStack` will cause a compilation error.

The result from the program in Figure 8.24 is that we have a generic Stack class where objects of different types can be manipulated in similar fashion. For instance, we only need a single method of `printStack` to print the content of `intStack,dobStack` and `chrStack` though the three of them have distinctive types.

Generic class is a powerful feature in Java that promotes software reusability. When functionality is type-independent, we can make the class type-generic so that the code is written once and can be applied to many.

8.5.2 Wild Card Method

Consider a simple method to print the content of an `ArrayList`. We can write our method (naively) as shown in Figure 8.29.

```
public static void printArray(ArrayList<Object> list)
{
        Iterator itr = list.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
}
```

Figure 8.29: Method to print the content of an `ArrayList`

This method works absolutely fine as long as we pass an `ArrayList` of type `Object` to the method. Consider the following code snippet shown in Figure 8.30.

```
Integer[] intNumbers = {1, 2, 3, 4, 5};
ArrayList <Integer> intList = new ArrayList<Integer>();
for(Integer element : intNumbers)
intList.add(element);
printArray(intList);
```

Figure 8.30: The code snippet of type `Object` to the method

The understanding so far, the above code snippet should work fine because Object is the super(super) class of Integer. However, when we compile the program, the compiler issues the error message as shown in Figure 8.31.

```
...printArray(java.util.ArrayList<java.lang.Object>) cannot be
pplied to (java.util.ArrayList<java.lang.Integer>)printArray(intList);
                                                  ^
```

Figure 8.31: The error message by compiler

The reason we get the error message is because the compiler does not consider the parameterised type ArrayList<Object> to be the supertype of ArrayList <Integer>, although in the class hierarchy, Object is the super(super) class of Integer. If it did, then every operation we can perform on ArrayList<Object> will also work on an ArrayList<Integer>. The use of supertype, Object actually has devastating effects to the method as we literally limit the method to work only with type Object. How do we overcome this problem? The answer is by the use of **wildcard** type argument. Wildcard arguments act as supertype of parameterised types. It is denoted by a question mark (?) literally means unknown type. The printArray method shown earlier can be rewritten using wildcard as shown in Figure 8.32.

```
public static void printArray(ArrayList<?> list)
{
        Iterator itr = list.iterator();
        while(itr.hasNext())
             System.out.println(itr.next());
}
```

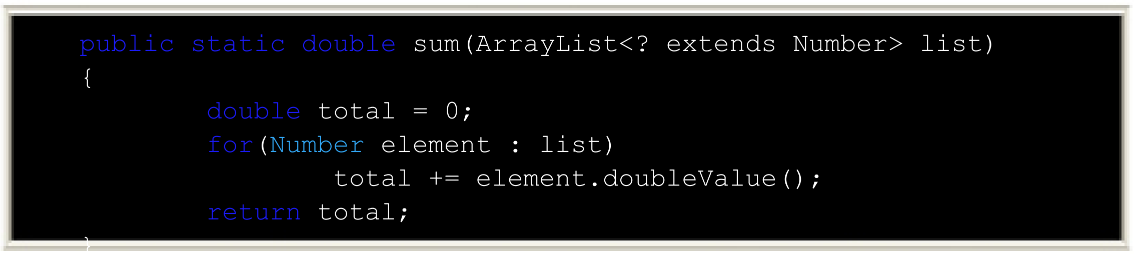
Figure 8.32: The printArray method can be rewritten using wildcard

With the wildcard argument, we solve our problem with Integer argument for method printArray as it can now accept argument of any type.

In some situations, we may want to limit the acceptable type to a method while maintaining the generality of the method with wildcards. For instance, a method that sums up the content of an ArrayList may want to ensure that it only accepts an ArrayList that is made up of numbers (i.e. any number). We can write:

```
ArrayList<? extends Number>
```

at the parameterised section of the generic. By doing so, we have limit the unknown type parameter to be the type of Number or any of its subtypes (Byte, Short, Integer, Double, etc). Number is the upper bound of the wildcard and this approach is known as bounded wildcard. The complete method for `sum` is shown in Figure 8.33.



```
public static double sum(ArrayList<? extends Number> list)
{
        double total = 0;
        for(Number element : list)
                total += element.doubleValue();
        return total;
```

Figure 8.33: The complete method for sum

If the wildcard is specified without an upper bound, only the method of type `Object` can be invoked on values of the wildcard type. Also, methods that use wildcards in their parameter's type argument cannot be used to add element to a collection referenced by the parameter.

SUMMARY

1. This chapter introduces the Java collection framework and Generics.
2. The Java collection framework provides a standard programming interface to many of the most common data structures such as:
   - `ArrayList`
   - `LinkedList`
   - `Queue`
   - `Stack`
3. These data structures are referred to as collection, which symbolise a group of objects that need to be operated upon together in some controlled fashion.
4. The collection framework demonstrates the benefit of code reused in object oriented programming where readymade well-crafted data structure algorithms can be shared and reused by others. This frees programmers from the arduous task of implementing the data structures, thus allowing programmers to devote their effort on the actual parts of their program than reinventing the wheel.
5. Generics is the new feature of Java (introduced in Java 5) that allow abstraction over types, which mean you can write methods or classes that is type independent. Although multi-typed methods has long been implemented using polymorphism, generics differ in the way that it ensure type-safe, which mean any incompatibility of type will be reported during compile-time where as polymorphism is not.

6. In addition, generics overcome the need to create multiple overloaded methods which will result in more manageable code.
7. Generics is utmost beneficial especially in the implementation of collections. It is not surprising that the collections we have presented used many of the generic capabilities.

GLOSSARY

Binary search	A search technique for sorted list by repeatedly dividing the search interval in half and comparing the middle element in the span to the target value.
Data Structure	Is a particular way of storing and organising data in a computer so that it can be manipulated efficiently.
Declarative	A programming paradigm that expresses what the program or function should accomplish, rather than describing how to accomplishing it.
Hash table	A data structure that uses a hash function to effectively map certain identifiers to certain value.
Linked list	A data structure consisting of a sequence of records (nodes) linked by memory references.
Stack	A data structure based on the principle of Last in first out (LIFO).
Synchronised	A process to ensure that the shared values of multiple threads are consistent with each other.
Thread	A sub-process that executes independently from the main process but share the resources (memory, state, address space, etc) with the main process.

MISCELLANEOUS QUESTIONS

1. ____________________ are classes that allow primitive type to be manipulated as objects.
2. Set, Queue and List have the parent class of ____________________.
3. ____________________ is a subclass of Set collection.
4. ____________________ perform operations on an entire collection.
5. The difference between List and Set is that a List can contain ____________________ element whereas a Set cannot.
6. A(n) ____________________ is used to walk through a collection.
7. Two of the concrete implementation of `List` interface are ____________________ and ____________________.
8. `ArrayList` is implemented using ____________________ object.
9. Vector is different from other collection the way that the `Vector` class is by default ____________________.
10. The `Stack` class in Java is implemented from ____________________ interface.
11. Map is a data structure where ____________________ are mapped to specific ____________________.
12. Three concrete implementation of Map interface are ____________________, ____________________ and ____________________.
13. A program code where type mismatch can be detected during compile time is said to be ____________________.
14. ____________________ allows programmers to write methods or classes that is type independent.
15. An unknown type argument in generic method is known as ____________________ type.

STRUCTURED QUESTIONS

Question 1

Briefly explain the operation of each of the following methods of `ArrayList`.

(a) `add`
(b) `addAll`
(c) `clear`
(d) `contain`
(e) `get`
(f) `indexOf`
(g) `remove`
(h) `set`

Question 2

Briefly explain the operation of each of the following methods of `HashMap`.

(a) `put`
(b) `get(c)`
(c) `containKey`
(d) `keySet`

Question 3

What is the difference between `Iterator` and `ListIterator`?

Question 4

Queue is a ________________________________ data structure.

Question 5

Stack is a ________________________________ data structure.

Question 6

SelectionSort is a method that sorts the elements in a collection given as argument. Write the method heading for the SelectionSort without using generic.

Question 7

Rewrite the method heading for SelectionSort in question 6 by using generic.

Question 8

Explain why the approach of writing the SelectionSort in question 7 is better than the approach in question 6.

MULTIPLE CHOICE QUESTIONS

1. Which is not a data structure of the Java collection framework?
   (a) ArrayList
   (b) TreeHashMap
   (c) BinaryTree
   (d) Stack
   (e) LinkedList

2. Which of the following is not the benefit of using Java collection framework?
   (a) Collection reduces programming effort.
   (b) Collection improves portability of the program.
   (c) Collection fosters software reuse.
   (d) Collection reduces effort to learn new APIs.
   (e) Collection makes your program become more user friendly.

3. What is the difference between set and list?
   (a) Set is not part of the Java collection framework whereas list is part of the framework.
   (b) Set stores elements in an unordered way whereas list is ordered.
   (c) Set cannot contain duplicate elements whereas list permits duplicate elements.
   (d) Set uses array as base data structure whereas list uses linked list as base data structure.
   (e) Set uses linked list as base data structure whereas list uses array as base data structure.

4. Which of the following statements is not true about `Vector` and `ArrayList`?
   (a) `Vector` is synchronised and `ArrayList` is not.
   (b) Both `Vector` and `ArrayList` implement dynamic resizable array as the base data structure.
   (c) `ArrayList` has no default size while `Vector` has a default size of 10.
   (d) Both implement the `List` interface.
   (e) Execution time for program using Vector is faster than `List` because `Vector` has more efficient data structure.

5. Which implementation of the List interface provides for the fastest insertion of a new element into the middle of the list?
   (a) `LinkedList`
   (b) `ArrayList`
   (c) `Vector`
   (d) None of the implementation above.
   (e) All of the implementation above.

6. What is the most accurate explanation of the following method heading declaration?

```
void printNumbers(Collection<Number> c)
```

   (a) The method `printNumbers` can print the collection c as long as of the type of the collection is a subclass of `Number`.
   (b) Passing an `ArrayList` of type `Integer` will not cause any problem to the program.
   (c) Passing an `ArrayList` of type `Integer` will cause compiler to throw an exception.
   (d) Passing an `ArrayList` of type `Integer` will not cause compiler to throw an exception, but exception will be thrown during runtime.
   (e) The syntax declaration of the code is erroneous.

7. What is the most accurate explanation of the following method heading declaration?

```
void printNumbers(Collection<? extends Number> c)
```

   (a) Passing an `ArrayList` of type `Object` will not cause any problem to the program.
   (b) Passing an `ArrayList` of type `Object` will not cause any problem to the program as long as the object can be casted to any subclass of Number.
   (c) Passing an `ArrayList` of type `Object` will cause compiler to throw an exception.
   (d) Passing an `ArrayList` of type `Object` will not cause compiler to throw an exception, but exception will be thrown during runtime.
   (e) The syntax declaration of the code is erroneous.

REFERENCES

Dietel, P. J., Dietel, H. M. (2007). *Java How to Program*, 7th Ed, Prentice Hall.

JavaTM SE 6 Platform at a Glance. Retrieved October 18, 2008 from website: http://java.sun.com/javase/6/docs/.

The Java Tutorial Retrieved October 18, 2008 from website: http://java.sun.com/docs/books/tutorial/index.html.

Weisfeld, M. (2008). *Object Oriented Thought Process 3rd Ed*, Addison-Wesley.