Data structures in Java: Linked List, Stacks, Queues, Sets, Maps; Generics: Generic classes and Type parameters, Implementing Generic Types, Generic Methods, Wrapper Classes, Concept of Serialization.

Linked List:

- A **linked list** is a data structure used for collecting a sequence of objects that allows efficient *addition* and *removal* of elements in the middle of the sequence.
- A Linked List structure the data in a way that minimizes the cost of insertion and removal.
- A linked list uses a sequence of *nodes*. Each node stores a value and a reference to the next node in the sequence (i.e. A linked list consists of a number of nodes, each of which has a reference to the next node.)
- When you insert a new node into a linked list, only the neighbouring node references need to be updated which is shown in below figure.

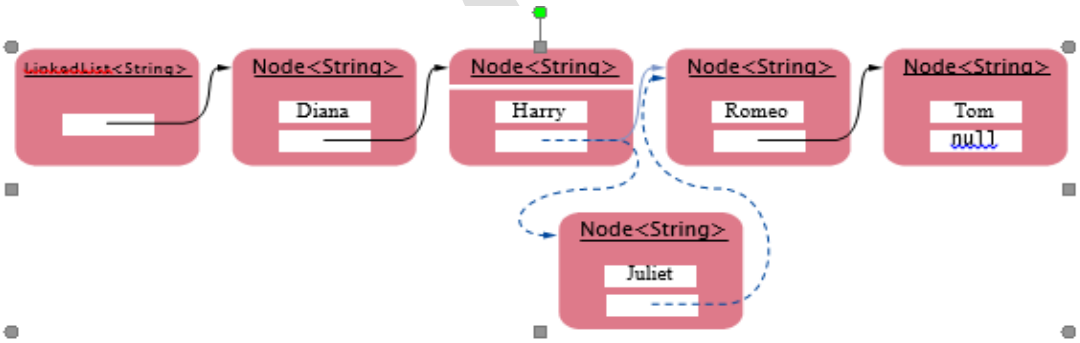


Figure 1 Inserting an Element into a Linked List

- Visiting the elements of a linked list in sequential order is efficient, but random access is inefficient.
- The Java library provides the `LinkedList` class in the `java.util` package is a **generic class (i.e.** We can specify the type of the list elements in angle brackets, such as `LinkedList<String>` or `LinkedList<Integer>`.)
- The methods shown in Table 1 give you direct access to the first and the last element in the `LinkedList`

Table 1 LinkedList Methods

`LinkedList<String> lst = new LinkedList<String>();`	An empty list.
`lst.addLast("Harry")`	Adds an element to the end of the list. Same as add.
`lst.addFirst("Sally")`	Adds an element to the beginning of the list. lst is now [Sally, Harry].
`lst.getFirst()`	Gets the element stored at the beginning of the list; here "Sally".
`lst.getLast()`	Gets the element stored at the end of the list; here "Harry".
`String removed = lst.removeFirst();`	Removes the first element of the list and returns it. removed is "Sally" and lst is [Harry]. Use removeLast to remove the last element.
`ListIterator<String> iter = lst.listIterator()`	Provides an iterator for visiting all list elements

- The Java library supplies a `ListIterator` class to add and remove elements in the middle of the list. A list iterator describes a position anywhere inside the linked list (see Figure 2).

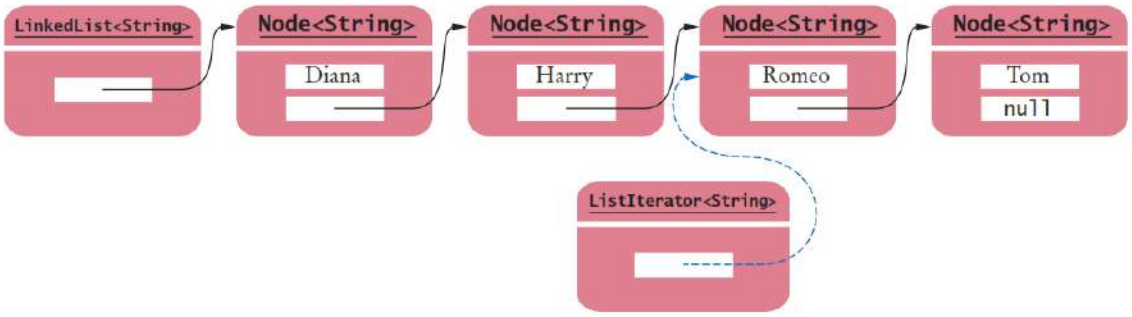


Figure 2 A List Iterator

- We can obtain a list iterator with the `listIterator()` method of the `LinkedList` class:

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator ();
```

- Initially, the iterator points before the first element. You can move the iterator position with the `next()` method:

```
iterator.next ();
```

- The next method throws a `NoSuchElementException` if you are already past the end of the list. You should always call the method `hasNext ()` before calling `next ()` —it returns true if there is a next element.

```
if (iterator.hasNext())
        iterator.next();
```

- The `next` method returns the element that the iterator is passing.
- We can traverse all elements in a linked list of strings with the following loop:

`while (iterator.hasNext())`
`{`
`String name = iterator.next();`
Do something with `name`
`}`
	`for (String name : employeeNames)`
	`{`
	Do something with `name`
	`}`

- The add method adds an object after the iterator, then moves the iterator position past the new element.

```
iterator.add("Juliet");
```

You can visualize insertion in the following figure:

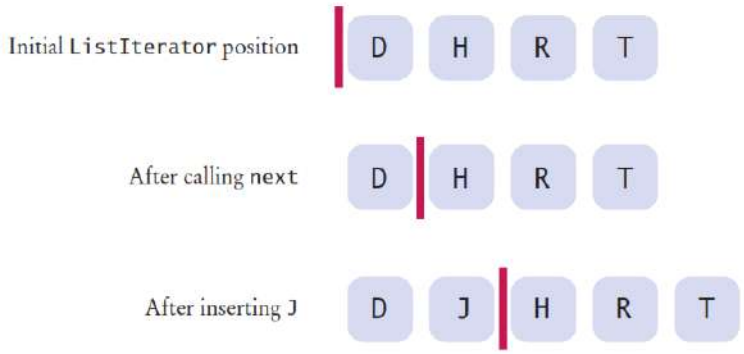


Figure 3 A Conceptual View of the List Iterator

- The nodes of the `LinkedList` class store two links: one to the next element and one to the previous one. Such a list is called a **doubly linked list**. You can use the `previous ()` and `hasPrevious ()` methods of the `ListIterator` interface to move the iterator position backwards.
- The `remove` method removes the object that was returned by the last call to `next` or `previous`. For example, the following loop removes all names that fulfil a certain condition.

```
while (iterator.hasNext())
{
        String name = iterator.next();
       if (name fulfills condition)
            iterator.remove();
}
```

- The remove () can be called only once after calling `next` or `previous`. The following is an error:

```
iterator.next();
iterator.next();
iterator.remove();
iterator.remove(); // Error: You cannot call remove twice.
```

- We cannot call remove () immediately after a call to add () :

```
iter.add("Fred");
iter.remove(); // Error: Can only call remove after calling next or previous
```

- If you call the `remove` method improperly, it throws an `IllegalStateException`
- Table 2 summarizes the methods of the `ListIterator` interface

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.
`if (iter.hasPrevious()) { s = iter.previous(); }`	`hasPrevious` returns `true` because the iterator is not at the beginning of the list.
`iter.add("Diana");`	Adds an element before the iterator position. The list is now `[Diana, Sally]`.
`iter.next(); iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is again `[Diana]`.

Example Program:

```
1. import java.util.LinkedList;
2. import java.util.ListIterator;

3. /**
4. This program demonstrates the LinkedList class.
5. */
6. public class ListDemo
7. {
8. public static void main(String[] args)
9. {
10. LinkedList<String> staff = new LinkedList<>();
11. staff.addLast("Diana");
12. staff.addLast("Harry");
13. staff.addLast("Romeo");
14. staff.addLast("Tom");

15. // | in the comments indicates the iterator position

16. ListIterator<String> iterator = staff.listIterator(); // |DHRT
17. iterator.next(); // D|HRT
18. iterator.next(); // DH|RT
```

```
// Add more elements after second element

iterator.add("Juliet"); // DHJ|RT
iterator.add("Nina"); // DHJN|RT

iterator.next(); // DHJNR|T

// Remove last traversed element

iterator.remove(); // DHJN|T

// Print all elements

System.out.println(staff);
System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
}
}
```

Stacks and Queues:

- A **stack** lets you insert and remove elements at only one end, traditionally called the *top* of the stack. To visualize a stack, think of a stack of books (see Figure 12).
- New items can be added to the top of the stack. Items are removed at the top of the stack as well.

Def. A stack is a collection of items with "last in, first out(LIFO)" retrieval

- A **queue** is similar to a stack, except that you add items to one end of the queue (the *tail)* and remove them from the other end of the queue (the *head).*

Def. A queue is a collection of items with "first in, first out (FIFO)" retrieval

- To visualize a queue, simply think of people lining up (see Figure)

Figure 12
A Stack of Books

- Queues store items in a *first in, first out* or *FIFO* fashion. Items are removed in the same order in which they have been added.
- There is a `Stack` class in the Java library that implements the abstract stack type and the `push` and `pop` operations
- The `Queue` interface in the standard Java library has methods add to add an element to the tail of the queue, `remove` to remove the head of the queue, and `peek` to get the head element of the queue without removing it.
- The `LinkedList` class also implements the `Queue` interface, and you can use it when a queue is required:

```
Queue<String> q = new LinkedList<String>();
```

- Table 4 shows how to use the stack and queue methods in Java.

Table 4 Working with Queues and Stacks

`Queue<Integer> q = new LinkedList<Integer>();`	The `LinkedList` class implements the `Queue` interface.
`q.add(1); q.add(2); q.add(3);`	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.
`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.
`Stack<Integer> s = new Stack<Integer>();`	Constructs an empty stack.
`s.push(1); s.push(2); s.push(3);`	Adds to the top of the stack; `s` is now `[1, 2, 3]`.
`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

- The `Stack` class in the Java library uses an array list to implement a stack
- A queue can be efficiently implemented as a linked list.

Example Program on Stack:

```
1. import java.util.Stack;
2. /**
3. This program simulates an undo stack. Note that operations
4. must be undone in the opposite order in which they are first
5. issued.
6. */
7. public class StackDemo
8. {
9. public static void main(String[] args)
10. {
11. Stack<String> commands = new Stack<>();
12. commands.push("Insert 'Hello'");
13. commands.push("Insert ','");
14. commands.push("Insert ' '");
15. commands.push("Insert 'World'");
16. commands.push("Insert '?'");
17. commands.push("Delete '?'");
18. commands.push("Insert '!'");
```

```
// Now we undo the last four commands
for (int i = 1; i <= 4; i++)
{
String command = commands.pop();
System.out.println("Undo " + command);
}
}
}
```

Example Program on Queue:

```
import java.util.LinkedList;
import java.util.Queue;
/**
This program simulates a print queue. Note that documents are printed
in the same order as they are submitted.
*/
public class QueueDemo
{
public static void main(String[] args)
{
Queue<String> jobs = new LinkedList<>();
jobs.add("Joe: Expense Report #1");
jobs.add("Cathy: Meeting Memo");
System.out.println("Printing " + jobs.remove());
jobs.add("Cathy: Purchase Order #1");
jobs.add("Joe: Expense Report #2");
jobs.add("Joe: Weekly Report");
System.out.println("Printing " + jobs.remove());
jobs.add("Cathy: Purchase Order #2");
while (jobs.size() > 0)
{
System.out.println("Printing " + jobs.remove());
}
}
}
```

Applications of queues and stacks in computer science:

- The Java graphical user interface system keeps an event queue of all events, such as mouse and key- board events. Events are removed and passed to event listeners in the order in which they were inserted
- A printer may be accessed by several applications, perhaps running on different computers, print jobs are printed using the "first in, first out" rule.
- Stacks are used when a "last in, first out" rule is required. For example, consider an algorithm that attempts to find a path through a maze.
- The **run-time stack** that a processor or virtual machine keeps to organize the variables of nested methods.
- **Reverse Polish Notation** that have the operators *follow* the operands use the stack data structure to evaluate the expression.

Sets:

Both Stack and Queue data structures keep the elements in the same order in which you inserted them. However, in many applications, you don't really care about the order of the elements in a collection.

Def. *A set is an unordered collection of distinct elements. Elements can be added, located, and removed.*

✓ The fundamental operations on a set:

- Adding an element
- Removing an element
- Locating an element (Does the set contain a given object?)
- Listing all elements (not necessarily in the order in which they were added)

✓ Sets don't have duplicates. Adding a duplicate of an element that is already present is silently ignored and attempting to remove an element that isn't in the set is silently ignored.

✓ There are two different data structures for implementing Set Operations, called *hash tables* and *trees*.

✓ The standard Java library provides set implementations based on both data structures, called `HashSet` and `TreeSet`. Both of these data structures implement the `Set` interface (see Figure 2).

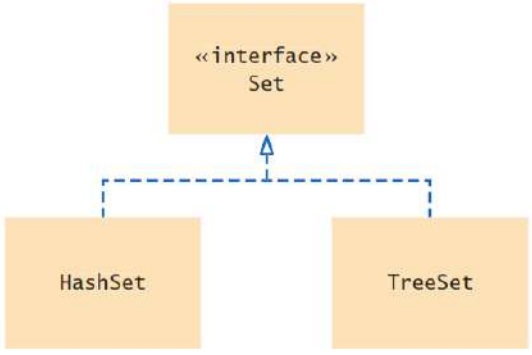


Figure 2
Set Classes and Interfaces in the Standard Library

- In order to use a `HashSet`, the elements must provide a `hashCode` method (for example `String`, `Integer`, `Point`, `Rectangle`, `Color`, and all the collection classes provide the `hashCode` method)

- Elements in a TreeSet are kept in sorted order. In order to use a `TreeSet`, the element type should implement the `Comparable` interface.
- As a rule of thumb, use a hash set unless you want to visit the set elements in sorted order.
- When you construct a `HashSet` or `TreeSet`, store the reference in a `Set<String>` vari- able, either as

```
Set<String> names = new HashSet<String>(); (or) Set<String> names = new TreeSet<String>();
```

- Adding and removing set elements are accomplished with the `add` and `remove` methods:

```
names.add("Romeo");
names.remove("Juliet");
```

- The `contains` method tests whether an element is contained in the set:

```
if (names.contains("Juliet")) . . .
```

- To list all elements in the set, we should use iterator

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
        String name = iter.next();
        Do something with name
}
```

The following table shows the list of methods and their useage :

Table 4 Working with Sets

`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<String>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.
`System.out.println(names);`	Prints the set in the format `[Fred, Romeo]`. The elements need not be shown in the order in which they were inserted.

`for (String name : names)`
`{`
`. . .`
`}`
	Use this loop to visit all elements of a set.

`names.remove("Romeo");`	Now `names.size()` is 1.
`names.remove("Juliet");`	It is not an error to remove an element that is not present. The method call has no effect.

Example Program:

The following program shows a practical application of sets. It reads in all words from a dictionary file that contains correctly spelled words and places them in a set. It then reads all words from a document—here, the book *Alice in Wonderland*—into a second set. Finally, it prints all words from that set that are not in the dictionary set. These are the potential misspellings.

1. import java.util.HashSet;

```
2. import java.util.Scanner;
3. import java.util.Set;
4. import java.io.File;
5. import java.io.FileNotFoundException;
6. /**
7. This program checks which words in a file are not present in a dictionary.
8. */
9. public class SpellCheck
10. {
11. public static void main(String[] args)
12. throws FileNotFoundException
13. {
14. // Read the dictionary and the document
15. Set<String> dictionaryWords = readWords("words");
16. Set<String> documentWords = readWords("alice30.txt");
17. // Print all words that are in the document but not the dictionary
18. for (String word : documentWords)
19. {
20. if (!dictionaryWords.contains(word))
21. {
22. System.out.println(word);
23. }
24. }
25. }
26. /**
27. Reads all words from a file.
28. @param filename the name of the file
29. @return a set with all lowercased words in the file. Here, a
30. word is a sequence of upper- and lowercase letters.
31. */
32. public static Set<String> readWords(String filename)
33. throws FileNotFoundException
34. {
35. Set<String> words = new HashSet<>();
36. Scanner in = new Scanner(new File(filename));
37. // Use any characters other than a-z or A-Z as delimiters
38. in.useDelimiter("[^a-zA-Z]+");
39. while (in.hasNext())
40. {
41. words.add(in.next().toLowerCase());
```

```
42. }
43. return words;
44. }
45. }
```

Sets:

- A **map** allows you to associate elements from a *key set* with elements from a *value collection.*
- Every key in the map has a unique value, but a value may be associated with several keys

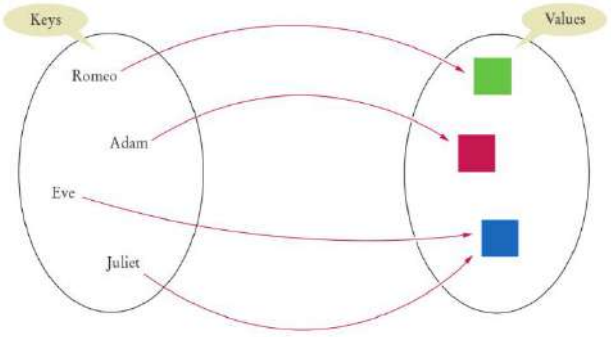


Figure 10 A Map

- The Java library has two implementations for maps: `HashMap` and `TreeMap`. Both of them implement the `Map` interface.
- Use a hash map unless you want to visit the keys in sorted order.
- After constructing a `HashMap` or `TreeMap`, you can store the reference to the map object in a `Map` reference

```
Map<String, Color> favoriteColors = new HashMap<String, Color>();

(or)

Map<String, Color> favoriteColors = new TreeMap<String, Color>();
```

- We can use the `put` method to add an association, we can change the value of an existing association, simply by calling `put` again and The `get` method returns the value associated with a key

```
favoriteColors.put("Juliet", Color.RED);
favoriteColors.put("Juliet", Color.BLUE);
Color julietsFavoriteColor = favoriteColors.get("Juliet");
```

- To remove a key and its associated value, use the `remove` method

```
favoriteColors.remove("Juliet");
```

- Sometimes you want to enumerate all keys in a map. The `keySet` method yields the set of keys

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{

Color value = m.get(key);
System.out.println(key + " : " + value);
```

```
}
```

- When you use a hash map, the keys are visited in a seemingly random order. With a tree map, keys are visited in sorted order.
- The following sample program shows a map in action.

```
1.  import java.awt.Color;
2.  import java.util.HashMap;
3.  import java.util.Map;
4.  import java.util.Set;

5.  /**
6.  This program demonstrates a map that maps names to colors.
7.  */
8.  public class MapDemo
9.  {
10. public static void main(String[] args)
11. {
12. Map<String, Color> favoriteColors = new HashMap<>();
13. favoriteColors.put("Juliet", Color.BLUE);
14. favoriteColors.put("Romeo", Color.GREEN);
15. favoriteColors.put("Adam", Color.RED);
16. favoriteColors.put("Eve", Color.BLUE);

17. // Print all keys and values in the map

18. Set<String> keySet = favoriteColors.keySet();
19. for (String key : keySet)
20. {
            Color value = favoriteColors.get(key);
            System.out.println(key + " : " + value);
21. }
22. }
23. }
```

Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are Integer wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<String, Integer>();`	Use a HashMap if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or null if the key is not present. n is 100, n2 is null.
`System.out.println(scores);`	Prints scores.toString(), a string of the form {Harry=90, Sally=100}

`for (String key : scores.keySet())`
`{`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

Generic classes and Type parameters

- **Generic programming is** the creation of programming constructs that can be used with many different types.
- In Java, generic programming can be achieved with **inheritance** or with **type parameters.**
- For example, `LinkedList` class achieves genericity by using ***inheritance*** whereas the Java library Class `ArrayList` class achieves genericity by using **type parameters.**
- A class with a **type parameter** that is used to specify the type of the objects that you want to store.
- A generic class has one or more type parameters.

Implementing Generic Types:

The syntax for declaring a generic class is given below

Syntax 17.1 Declaring a Generic Class

Syntax

```
accessSpecifier class GenericClassName<TypeVariable1, TypeVariable2, . . .>
{
   instance variables
   constructors
   methods
}
```

Example

```
public class Pair<T, S>
{
   private T first;
   private S second;
   . . .
   public T getFirst() { return first; }
   . . .
}
```

Supply a variable for each type parameter.

Instance variables with a variable data type

A method with a variable return type

- When declaring a generic class, you specify a type variable for each type parameter. In the above example T,S are type variables.
- In general, it is customary to use short, uppercase names for type parameters such as the following

Type Variable	Meaning
E	Element type in a collection
K	Key type in a map
V	Value type in a map
T	General type
S, U	Additional general types

- Type variables of a generic class follow the class name and are enclosed in angle brackets.
- We should place the type variables for a generic class after the class name, enclosed in angle brackets (< and >).

```
e.g, public class Pair<T, S>
```

- When you declare the instance variables and methods of the `Pair` class, use the variable `T` for the first element type and `S` for the second element type

```
1.  public class Pair<T, S>
2.  {

3.     private T first;
4.     private S second;

5.     public Pair(T firstElement, S secondElement)
6.     {
7.        first = firstElement;
8.        second = secondElement;
9.     }
10.    public T getFirst() { return first; }
11.    public S getSecond() { return second; }
12. }
```

- In order to use a generic class, you need to *instantiate* the type parameter, that is, supply an actual type. You

can supply any class or interface type, for example

```
Pair<String, Integer> result = new Pair<String, Integer>("Harry Morgan", 1729);
```

- When you instantiate a generic class, the type that you supply replaces all occurrences of the type variable in the declaration of the class.
- The following sample program shows how to make use of a `Pair` for returning two values from a method.

PairDemo.java

```
1.  public class PairDemo
2.  {
3.  public static void main(String[] args)
4.  {
5.  String[] names = { "Tom", "Diana", "Harry" };
6.  Pair<String, Integer> result = firstContaining(names, "a");
7.  System.out.println(result.getFirst());
8.  System.out.println("Expected: Diana");
9.  System.out.println(result.getSecond());
10. System.out.println("Expected: 1");
11. }

12. /**
13. Gets the first String containing a given string, together
14. with its index.
15. @param strings an array of strings
16. @param sub a string
17. @return a pair (strings[i], i) where strings[i] is the first
18. strings[i] containing str, or a pair (null, -1) if there is no
19. match.
20. */
21. public static Pair<String, Integer> firstContaining(
22. String[] strings, String sub)
23. {
24. for (int i = 0; i < strings.length; i++)
25. {
            if (strings[i].contains(sub))
            {
                i.  return new Pair<>(strings[i], i);
            }
26. }
27. return new Pair<>(null, -1);
28. }
29. }
```

Pair.java

```
1.  /**
2.  This class collects a pair of elements of different types.
3.  */
4.  public class Pair<T, S>
5.  {
6.  private T first;
7.  private S second;
```

```
8.  /**
9.  Constructs a pair containing two given elements.
10. @param firstElement the first element
11. @param secondElement the second element
12. */
13. public Pair(T firstElement, S secondElement)
14. {
15. first = firstElement;
16. second = secondElement;
17. }

18. /**
19. Gets the first element of this pair.
20. @return the first element
21. */
22. public T getFirst() { return first; }

23. /**
24. Gets the second element of this pair.
25. @return the second element
26. */
27. public S getSecond() { return second; }

28. public String toString() { return "(" + first + ", " + second + ")"; }
29. }
```

Advantages of type parameters over inheritance:

- The `ArrayList` class, using the type variable `E` for the element type is given below

```
public class ArrayList<E>
{
public ArrayList() { . . . }
public void add(E element) { . . . }
. . .
}
```

- In order to use a generic class, you need to *instantiate* the type parameter, that is, supply an actual type.

  e.g. ArrayList<BankAccount>

  ArrayList<Measurable>

- The add() method for ArrayList<BankAccount> has the type variable E replaced with the type BankAccount:

  public void add(BankAccount element)

- The add method of the LinkedList class use inheritance to implement genericity

  public void add(Object element)

- The add method of the generic ArrayList class is safer. It is impossible to add a String object into an ArrayList<BankAccount>, but you can accidentally add a String into a LinkedList that is intended to hold bank accounts.

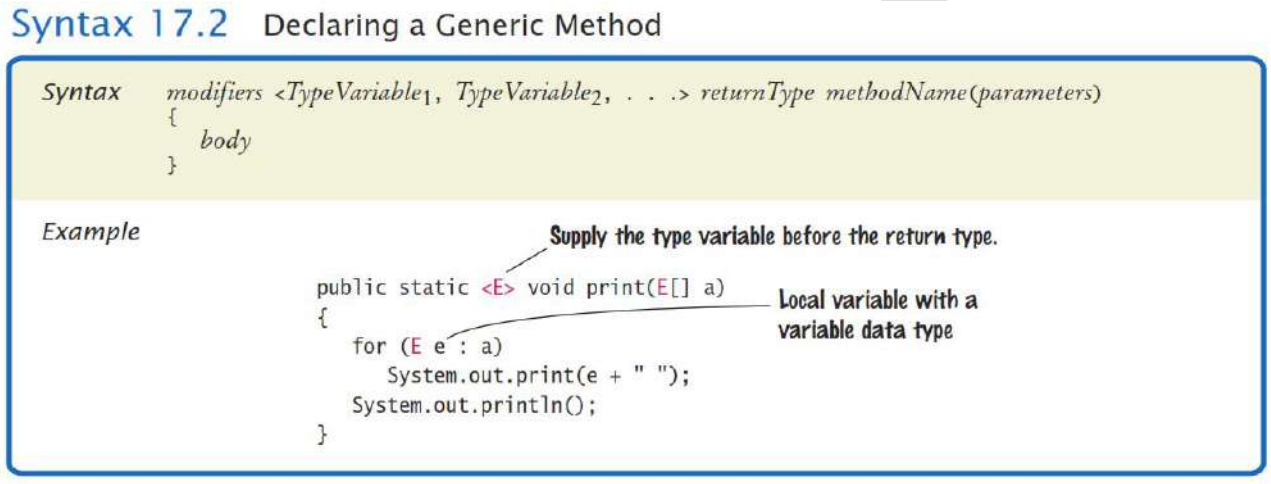
```
ArrayList<BankAccount> accounts1 = new ArrayList<BankAccount>();
LinkedList accounts2 = new LinkedList(); // Should hold BankAccount objects
accounts1.add("my savings"); // Compile-time error
accounts2.add("my savings"); // Not  detected at compile time
```

- The latter will result in a class cast exception when some other part of the code retrieves the string, believing it to be a bank account:

```
BankAccount account = (BankAccount) accounts2.getFirst(); // Run-time error
```

Generic Methods

- A generic method is a method with a type parameter. Generic method can occur in a class that in itself is not generic.
- We can think of it as a template for a set of methods that differ only by one or more types.
- The syntax for declaring a Generic method is given below



- In the above example, we declare a method that can print an array of any type
- It is often easier to see how to implement a generic method by starting with a concrete example. This method prints the elements in an array of strings.

```
1. public class ArrayUtil
2. {
3. public static void print(String[] a)
4. {
5. for (String e : a) System.out.print(e + " ");
6. System.out.println();
7. }
8. . . .
9. }
```

- In order to make the method into a generic method, replace String with a type parameter, say E, to denote the element type of the array.
- Add a type parameter list, enclosed in angle brackets, between the modifiers (public static) and the return type (void):

```
public static <E> void print(E[] a)
```

```
{
 for (E e : a)
        System.out.print(e + "\t");
System.out.println();
}
```

Wrapper Classes

- To treat primitive type values as objects, you must use wrapper classes.
- To store sequences of numbers in an array list, you must turn them into objects by using **wrapper classes.**
- There are wrapper classes for all eight primitive types

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

- Each wrapper class object contains a value of the corresponding primitive type. For example, an object of the class `Double` contains a value of type `double` (see Figure).

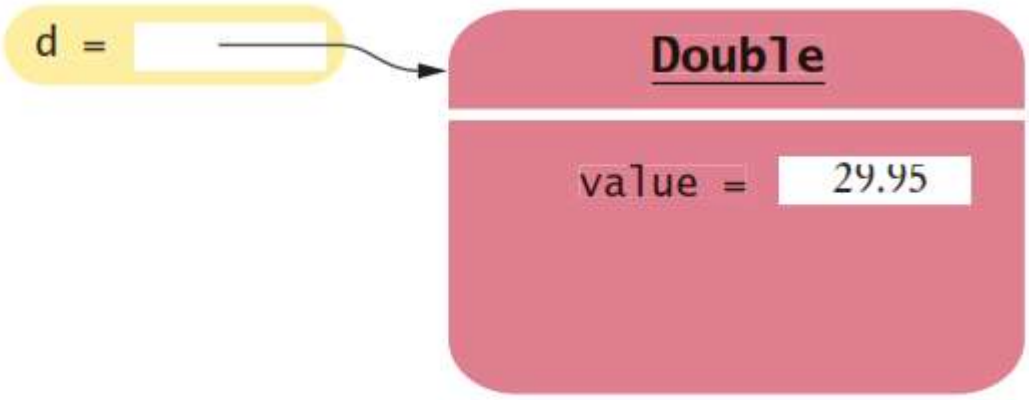


Figure 7 An Object of a Wrapper Class

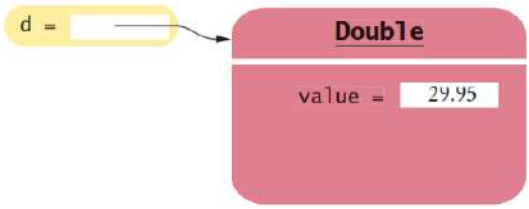


Figure 7 An Object of a Wrapper Class

- Conversion between primitive types and the corresponding wrapper classes is automatic. This process is called auto-boxing.
  For example, if you assign a number to a Double object, the number is automatically "put into a box", namely a wrapper object.
  Double d = 29.95; // Auto-boxing;
  same as Double d = new Double (29.95);
- Conversely, wrapper objects are automatically "unboxed" to primitive types.
  double x = d; // Auto-unboxing; same as double x = d.doubleValue();

  Auto-boxing even works inside arithmetic expressions. For example, the statement
  d = d + 1;
  is perfectly legal. It means:

  • Auto-unbox **d** into a double
  • Add 1
  • Auto-box the result into a new Double
  • Store a reference to the newly created wrapper object in d

  In order to collect numbers in an array list, simply remember to use the wrapper type as the type parameter, and then rely on auto-boxing.
  ArrayList<Double> values = new ArrayList<Double>();
  values.add(29.95);
  double x = values.get(0);

- Storing wrapped numbers is quite inefficient for long sequences of numbers or characters.

Concept of Serialization:

- The process of saving objects to a stream is called **serialization** because each object is assigned a serial number on the stream.
- To place objects of a particular class into an object stream, the class must implement the **Serializable** interface and the interface had no methods.
  class BankAccount implements Serializable

{

…

}

- We can use object streams to save and restore all instance variables of an object automatically.
- The `ObjectOutputStream` class can save entire objects out to disk, and the `ObjectInputStream` class can read them back in.
- For example, you can write a BankAccount object to a file as follows:

```
BankAccount b = . . .;

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("bank.dat"));

out.writeObject(b);
```

- The object output stream automatically saves all instance variables of the object to the stream.
- When reading the object back in, you use the `readObject` method of the `ObjectInputStream` class.
- That method returns an `Object` reference, so we need to remember the types of the objects that you saved and use a cast

  ObjectInputStream in = new ObjectInputStream(new FileInputStream("bank.dat"));
  BankAccount b = (BankAccount) in.readObject();
- The `readObject` method can throw a `ClassNotFoundException`—it is a checked exception, so you need to catch or declare it.
- The following is a sample program that puts serialization to work.

SerialDemo.java

```
1. import java.io.File;
2. import java.io.IOException;
3. import java.io.FileInputStream;
4. import java.io.FileOutputStream;
5. import java.io.ObjectInputStream;
6. import java.io.ObjectOutputStream;


7. /**
8. This program demonstrates serialization of a Bank object.
9. If a file with serialized data exists, then it is loaded.
10. Otherwise the program starts with a new bank.
11. Bank accounts are added to the bank. Then the bank
12. object is saved.
13. */
14. public class SerialDemo
15. {
16. public static void main(String[] args)
```

```
throws IOException, ClassNotFoundException
{
Bank firstBankOfJava;

File f = new File("bank.dat");
if (f.exists())
{
try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(f)))
{
firstBankOfJava = (Bank) in.readObject();
}
}
else
{
firstBankOfJava = new Bank();
firstBankOfJava.addAccount(new BankAccount(1001, 20000));
firstBankOfJava.addAccount(new BankAccount(1015, 10000));
}

// Deposit some money
BankAccount a = firstBankOfJava.find(1001);
a.deposit(100);
System.out.println(a.getAccountNumber() + ": " + a.getBalance());
a = firstBankOfJava.find(1015);
System.out.println(a.getAccountNumber() + ": " + a.getBalance());

try (ObjectOutputStream out = new ObjectOutputStream(
new FileOutputStream(f)))
{
out.writeObject(firstBankOfJava);
}
}
}
```

BankAccount.java

```
import java.io.Serializable;
/* A bank account has a balance that can be changed by deposits and withdrawals.*/
public class BankAccount implements Serializable
{
private int accountNumber;
```

```
6.  private double balance;
7.  /*          Constructs a bank account with a zero balance.
        @param anAccountNumber the account number for this account          */
8.  public BankAccount(int anAccountNumber)
9.  {
10. accountNumber = anAccountNumber;
11. balance = 0;
12. }
13. /**
14. Constructs a bank account with a given balance.
15. @param anAccountNumber the account number for this account
16. @param initialBalance the initial balance
17. */
18. public BankAccount(int anAccountNumber, double initialBalance)
19. {
20. accountNumber = anAccountNumber;
21. balance = initialBalance;
22. }
23. /**
24. Gets the account number of this bank account.
25. @return the account number
26. */

27. public int getAccountNumber()
28. {
29. return accountNumber;
30. }
31. /**
32. Deposits money into the bank account.
33. @param amount the amount to deposit
34. */
35. public void deposit(double amount)
36. {
37. double newBalance = balance + amount;
38. balance = newBalance;
39. }
40. /**
41. Withdraws money from the bank account.
42. @param amount the amount to withdraw
43. */
```

```
44. public void withdraw(double amount)
45. {
46. double newBalance = balance - amount;
47. balance = newBalance;
48. }
49. /**
50. Gets the current balance of the bank account.
51. @return the current balance
52. */
53. public double getBalance()
54. {
55. return balance;
56. }
57. }
```

Bank.java

```
1. import java.io.Serializable;
2. import java.util.ArrayList;
3. /** This bank contains a collection of bank accounts.  */
4. public class Bank implements Serializable
5. {
6. private ArrayList<BankAccount> accounts;
7. /*  Constructs a bank with no bank accounts.  */
8. public Bank()
9. {
10. accounts = new ArrayList<>();
11. }
12. /*
13. Adds an account to this bank.
14. @param a the account to add
15. */
16. public void addAccount(BankAccount a)
17. {
18. accounts.add(a);
19. }

20. /**
```

```
21. Finds a bank account with a given number.
22. @param accountNumber the number to find
23. @return the account with the given number, or null if there
24. is no such account
25. */
26. public BankAccount find(int accountNumber)
27. {
28. for (BankAccount a : accounts)
29. {
30. if (a.getAccountNumber() == accountNumber) // Found a match
31. {
        a. return a;
32. }
33. }
34. return null; // No match in the entire array list
35. }
36. }
```