Unit 9: Binary I/O, Recursion and Generics

Ravikumar R N
Assistant Professor
Computer Engineering
Marwadi University

Content

- Text I/O and Binary I/O
- Binary I/O classes
- Object I/O
- Random access File

Text I/O and Binary I/O

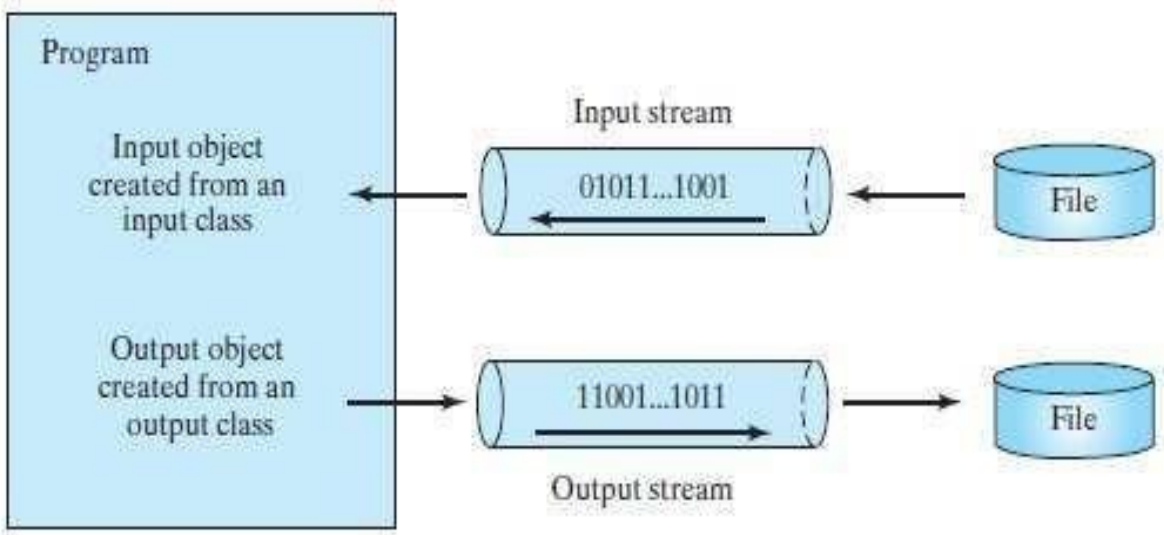
- Files can be classified as either **text** or **binary**.
- A file that can be processed (read, created, or modified) using a text editor such as Notepad, is called a **text file**.
- You cannot read binary files using a text editor, they are designed to be read by programs.
- Ex. Java file
- Java provides many classes for performing text I/O and binary I/O.

Text I/O and Binary I/O

- Text file - sequence of characters
- **binary file - sequence of bits**
- Ex. decimal integer 199 is stored as a sequence of three characters 1, 9, 9 in a text file, and the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals hex C7.
- **The advantage of binary files is that they are more efficient to process than text files.**
- Java offers many classes for performing file input and output. These can be categorized as ***text I/O** classes and **binary I/O** classes*.

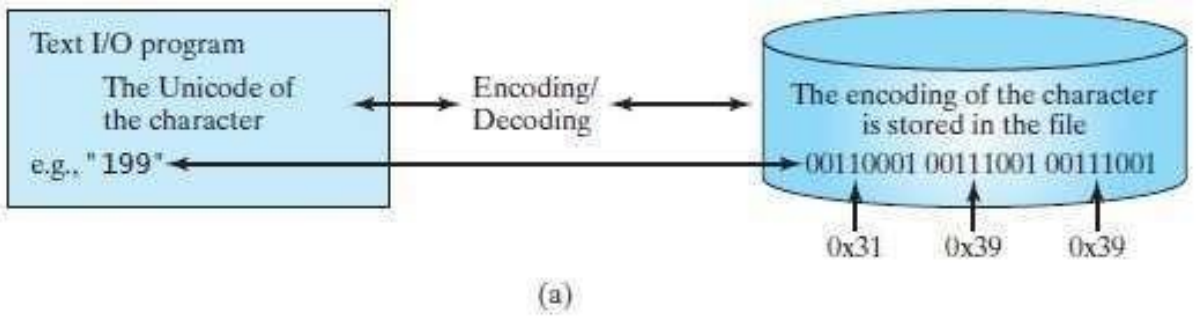
Text I/O

- Text data are read using the **Scanner** class and written using the **PrintWriter** class.
- An **input** class contains the methods to **read** data, and an **output** class contains the methods to **write** data.

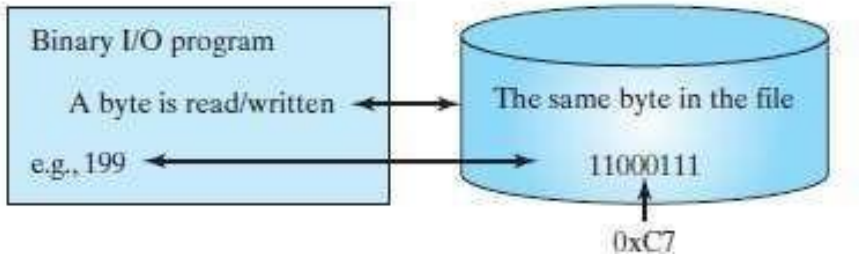


Text I/O vs. Binary I/O

- Binary I/O does not involve encoding or decoding and thus is more efficient than text I/O.
- **All the files in a computer are stored in binary format.**
- Encoding and decoding are automatically performed for text I/O.



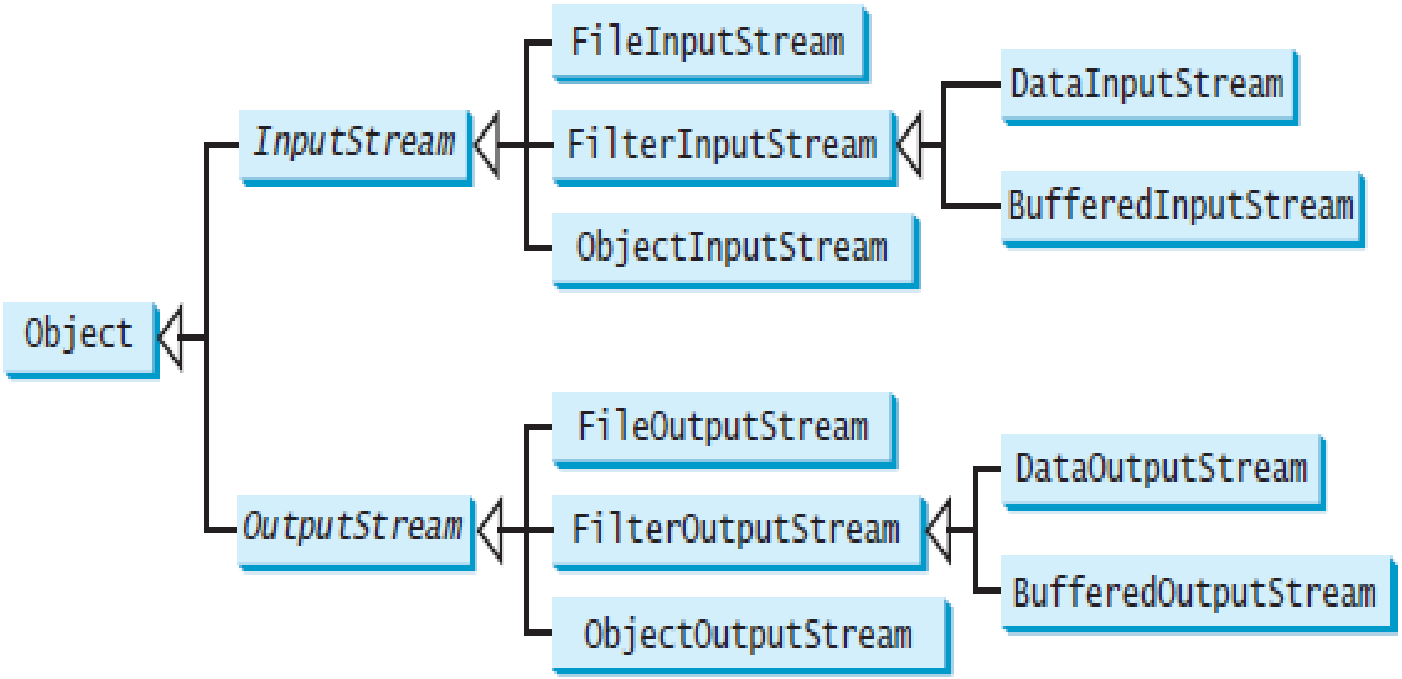
(a)



Text I/O vs. Binary I/O

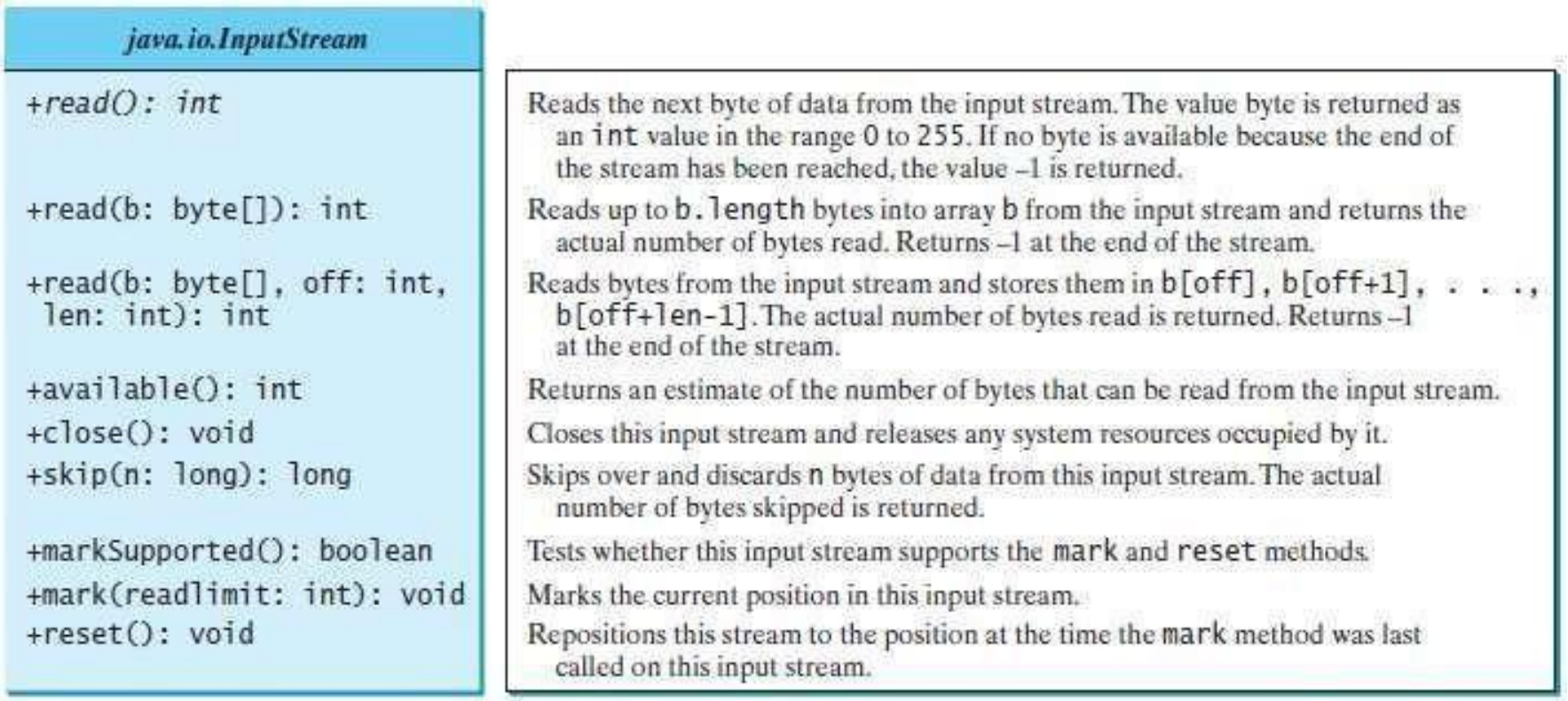
- Binary I/O does not require conversions.
- Binary files are independent of the encoding scheme on the host machine and thus are portable.
- Java programs on any machine can read a binary file created by a Java program. This is why Java class files are binary files. Java class files can run on a JVM on any machine.

Binary I/O Classes

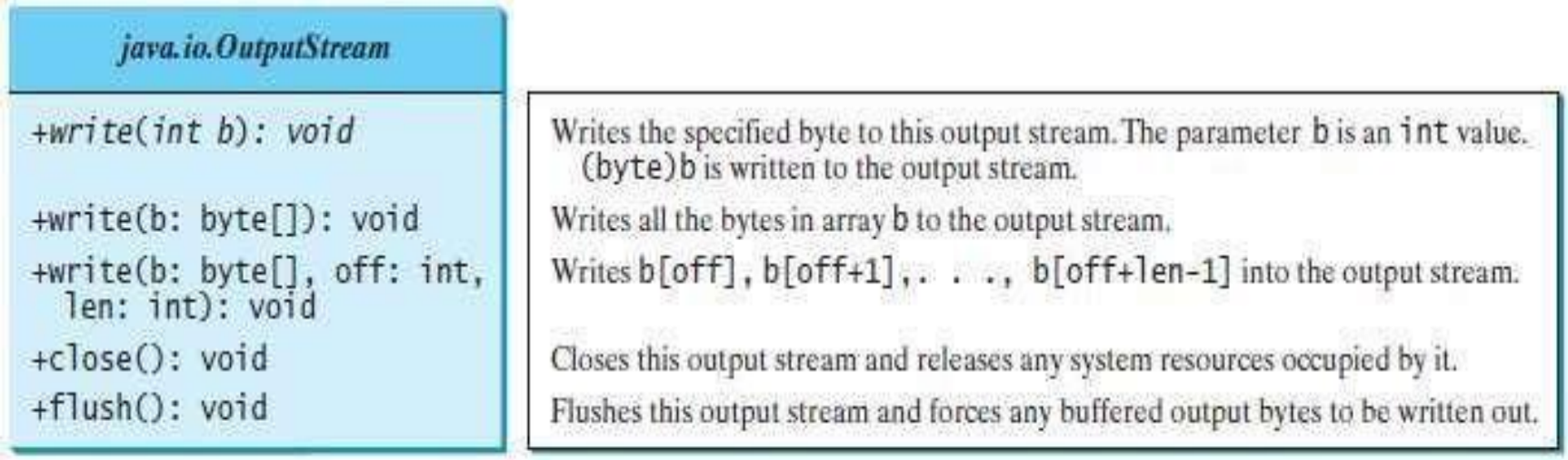


- The abstract InputStream is the root class for reading binary data, and the abstract OutputStream is the root class for writing binary data.

InputStream class

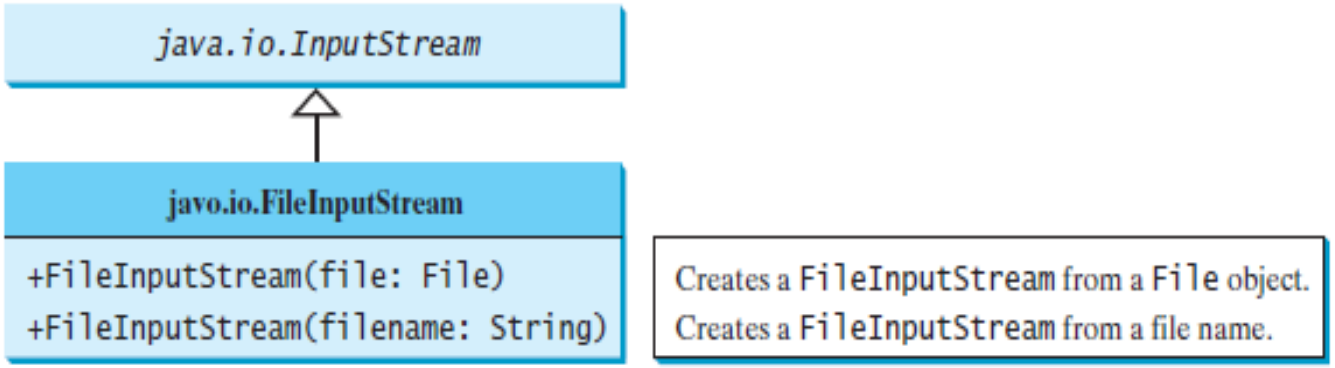


OutputStream class



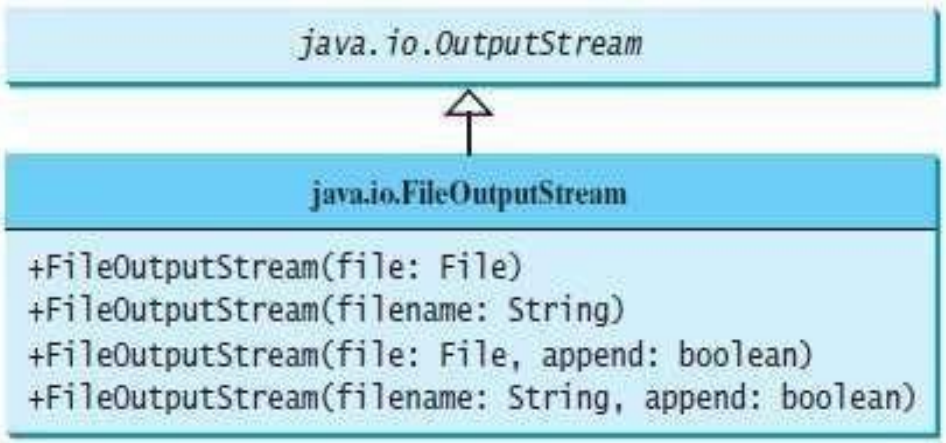
FileInputStream / FileOutputStream

- All the methods in these classes are inherited from InputStream and OutputStream.
- A java.io.FileNotFoundException will occur if you attempt to create a FileInputStream with a nonexistent file.



FileInputStream / FileOutputStream

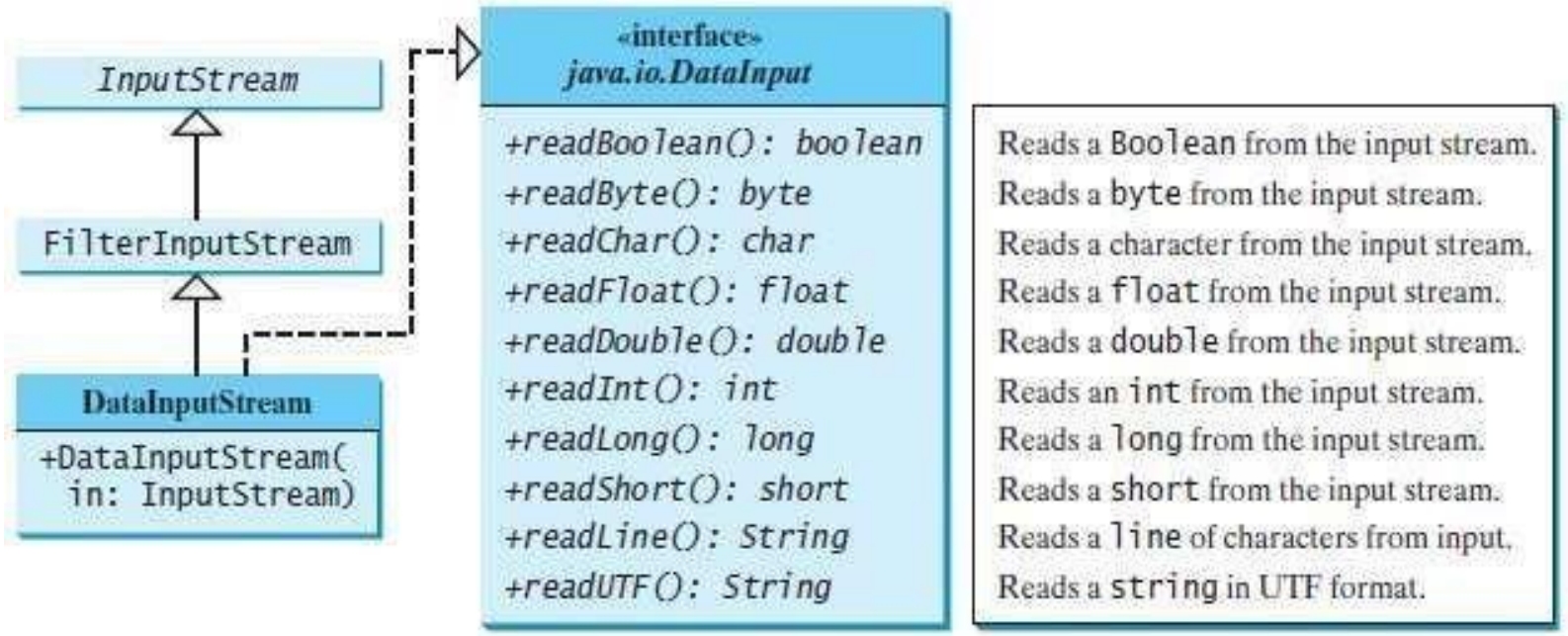
- If the file does not exist, a new file will be created. If the file already exists, the first two constructors will delete the current content of the file.
- To retain the current content and append new data into the file, use the last two constructors and pass true to the append parameter.



Creates a `FileOutputStream` from a `File` object.
Creates a `FileOutputStream` from a file name.
If `append` is true, data are appended to the existing file.
If `append` is true, data are appended to the existing file.

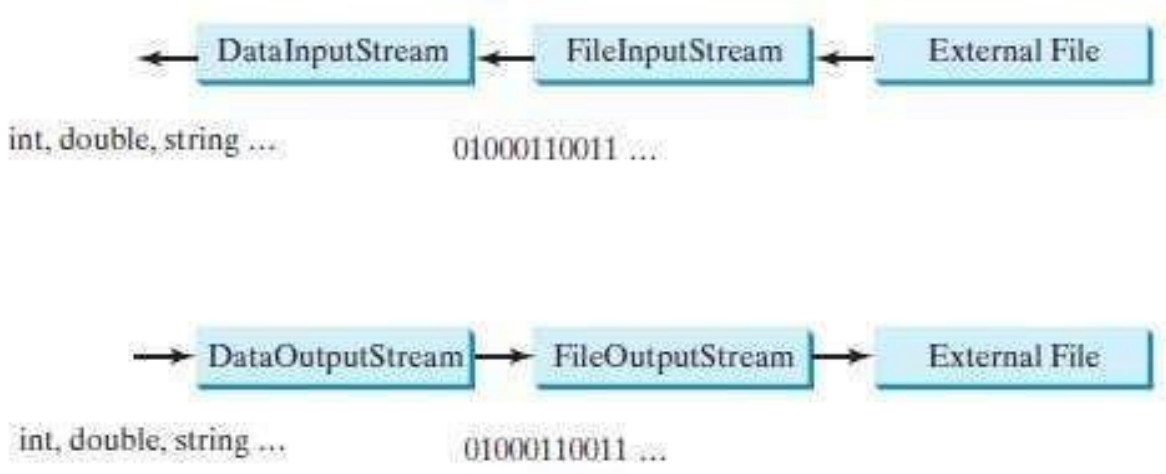
DataInputStream/ DataOutputStream

- DataInputStream reads bytes from the stream and converts them into appropriate primitive-type values or strings.
- DataOutputStream converts primitive-type values or strings into bytes and outputs the bytes to the stream.



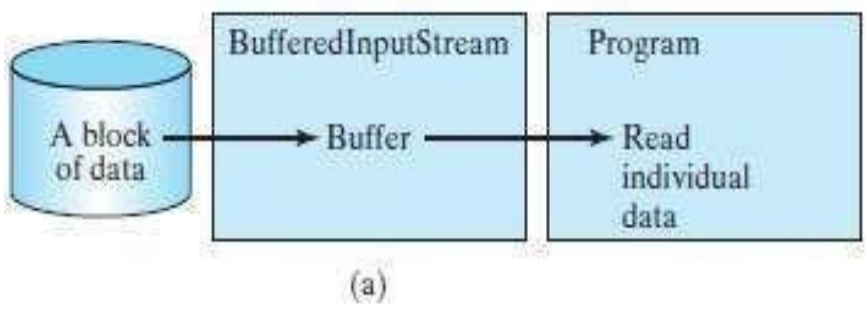
DataInputStream/DataOutputStream

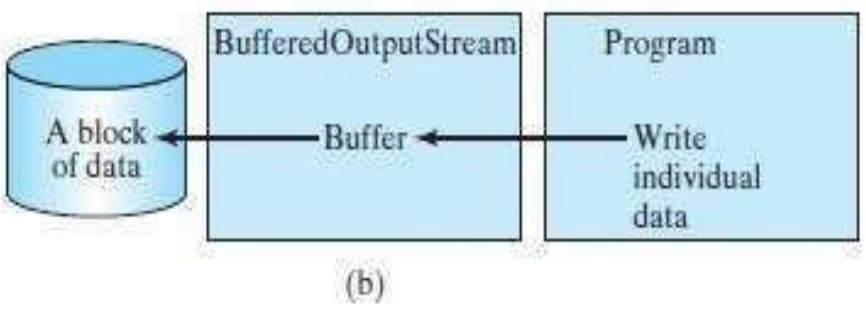
- **DataInputStream** and **DataOutputStream** read and write Java primitive-type values and strings in a machine-independent fashion, thereby enabling you to write a data file on one machine and read it on another machine that has a different operating system or file structure.



BufferedInputStream/ BufferedOutputStream

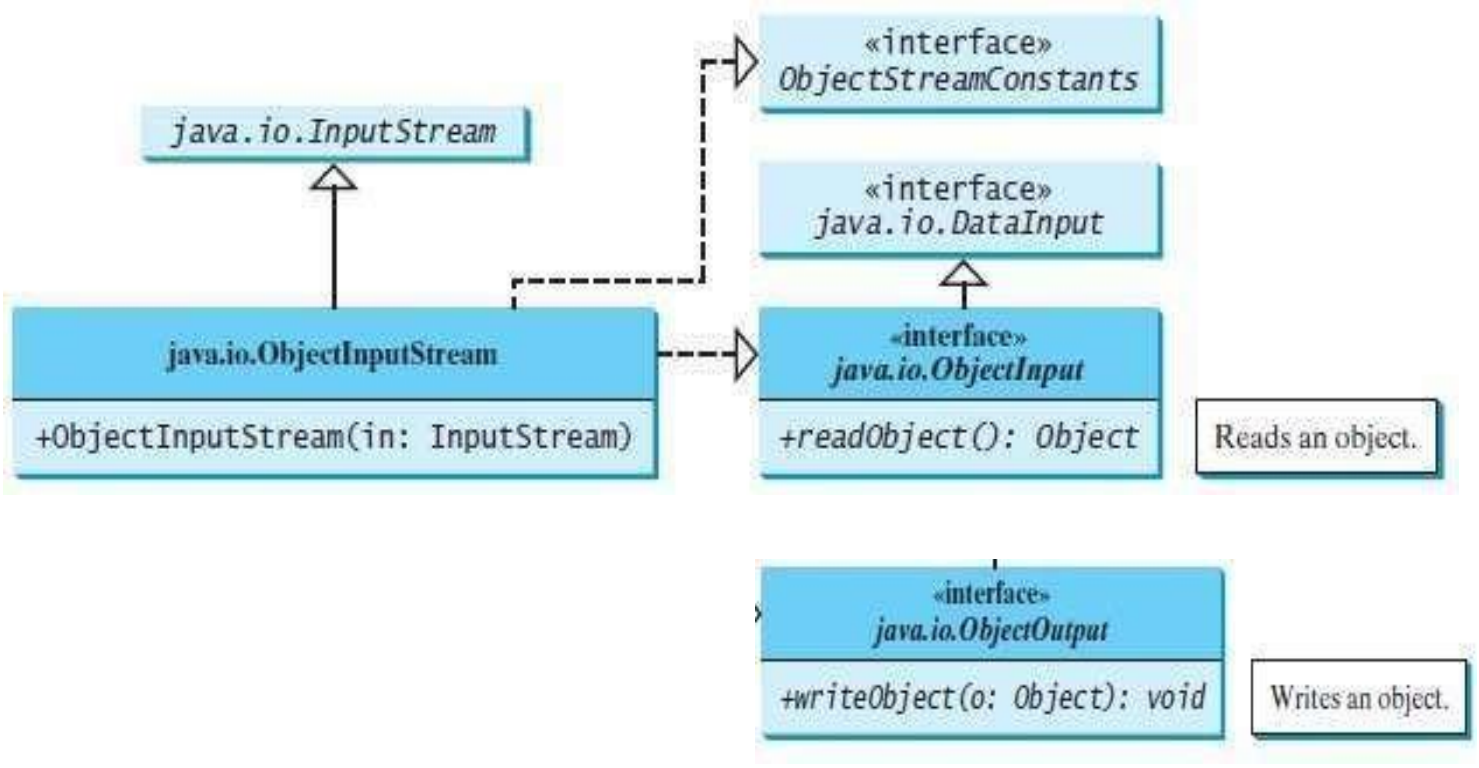
- BufferedInputStream/BufferedOutputStream can be used to speed up input and output by reducing the number of disk reads and writes.





Object I/O

- ObjectInputStream/ ObjectOutputStream classes can be used to read/write serializable objects.



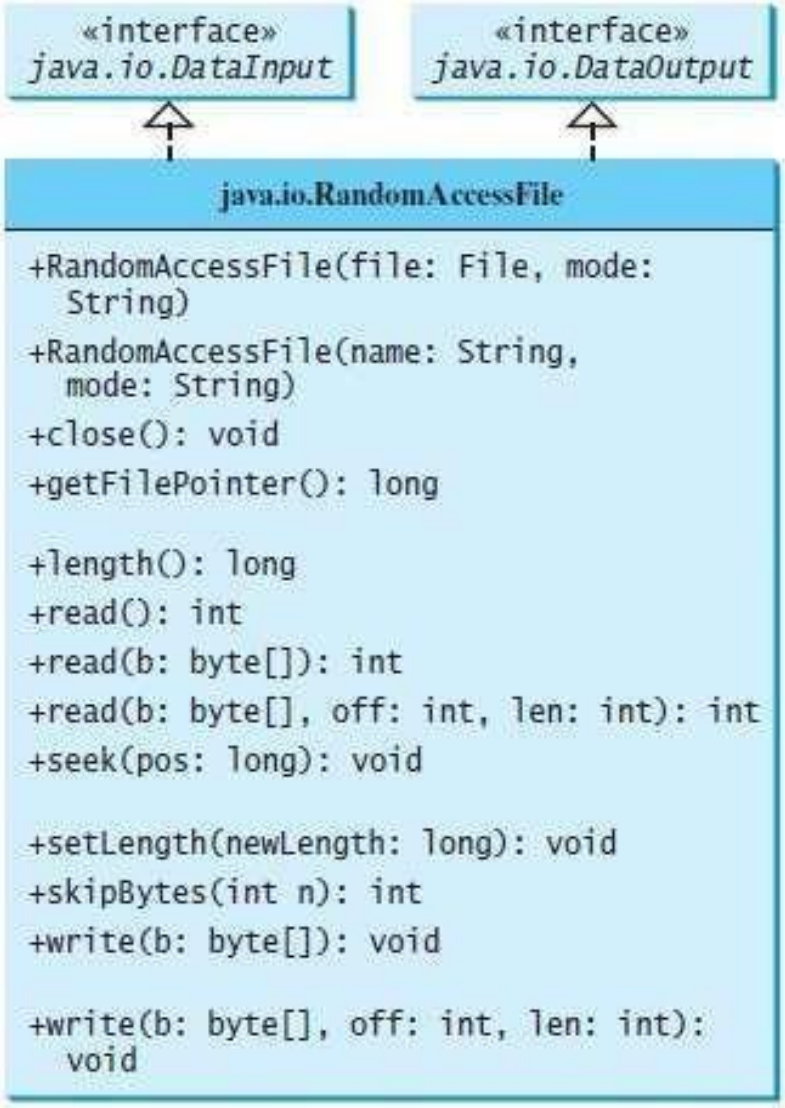
The Serializable Interface

- A serializable object is an instance of the java.io.Serializable interface, so the object's class must implement Serializable.
- Serialization in Java is a mechanism of writing the state of an object into a byte-stream.
- The Serializable interface is a **marker interface**. Since it **has no methods**, you don't need to add additional code in your class that implements Serializable.
- Java provides a built-in mechanism to automate the process of writing objects. This process is referred as **object serialization**, *which is implemented in* ObjectOutputStream.

Random-Access Files

- Java provides the RandomAccessFile class to allow data to be read from and written to at any locations in the file.
- All of the streams you have used so far are known as **read-only** or **write-only streams**. These streams are called **sequential streams**.

- Java provides the **RandomAccessFile** class to **allow data to be read from and written to at any locations in a file**. A file that is opened using the RandomAccessFile class is known as a **random-access file**.
- The RandomAccessFile class implements the **DataInput** and **DataOutput** interfaces.
- When creating a RandomAccessFile, you can specify one of two modes: **r or rw**. Mode r means that the stream is **read-only**, and mode rw indicates that the stream allows **both read and write**.



Creates a `RandomAccessFile` stream with the specified `File` object and mode.

Creates a `RandomAccessFile` stream with the specified file name string and mode.

Closes the stream and releases the resource associated with it.

Returns the offset, in bytes, from the beginning of the file to where the next `read` or `write` occurs.

Returns the number of bytes in this file.

Reads a byte of data from this file and returns –1 at the end of stream.

Reads up to `b.length` bytes of data from this file into an array of bytes.

Reads up to `len` bytes of data from this file into an array of bytes.

Sets the offset (in bytes specified in `pos`) from the beginning of the stream to where the next `read` or `write` occurs.

Sets a new length for this file.

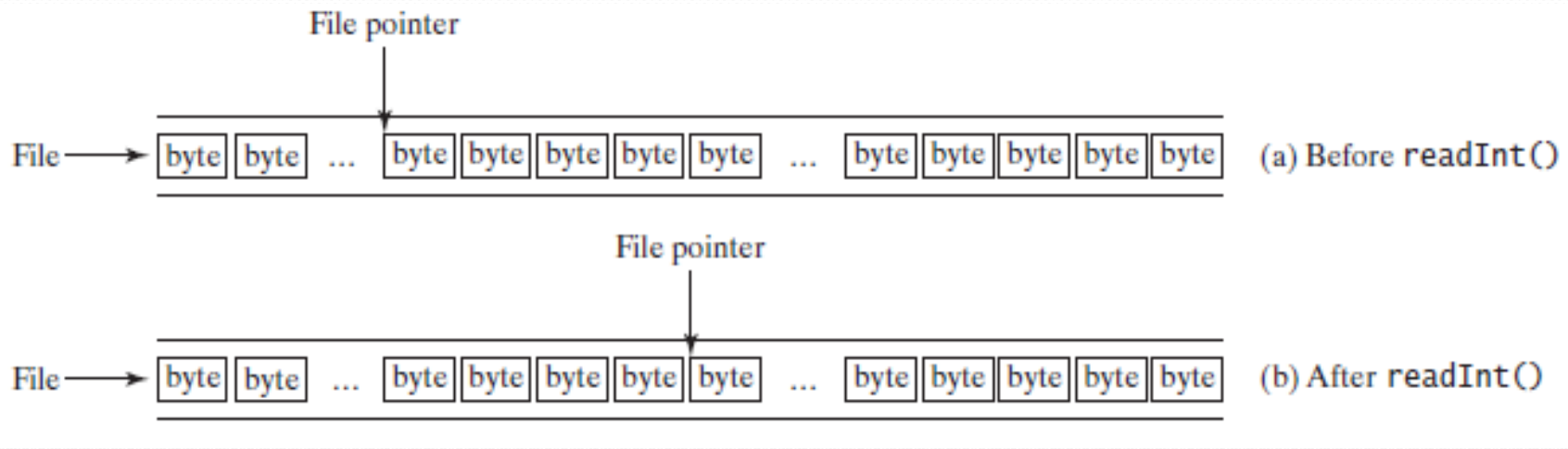
Skips over `n` bytes of input.

Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.

Writes `len` bytes from the specified byte array, starting at offset `off`, to this file.

Random-Access Files

- A random-access file consists of a sequence of bytes. A special marker called a **file pointer** is positioned at one of these bytes.
- For example, if you read an int value using readInt(), the JVM reads 4 bytes from the file pointer.



Content

- Recursion
- Problem Solving using Recursion
- Recursive Helper Methods
- Recursion vs. Iteration
- Tail Recursion

Recursion

- Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.
- In some cases, it enables you to develop a natural, straightforward, simple solution to an otherwise difficult problem.
- A recursive method is one that invokes itself.

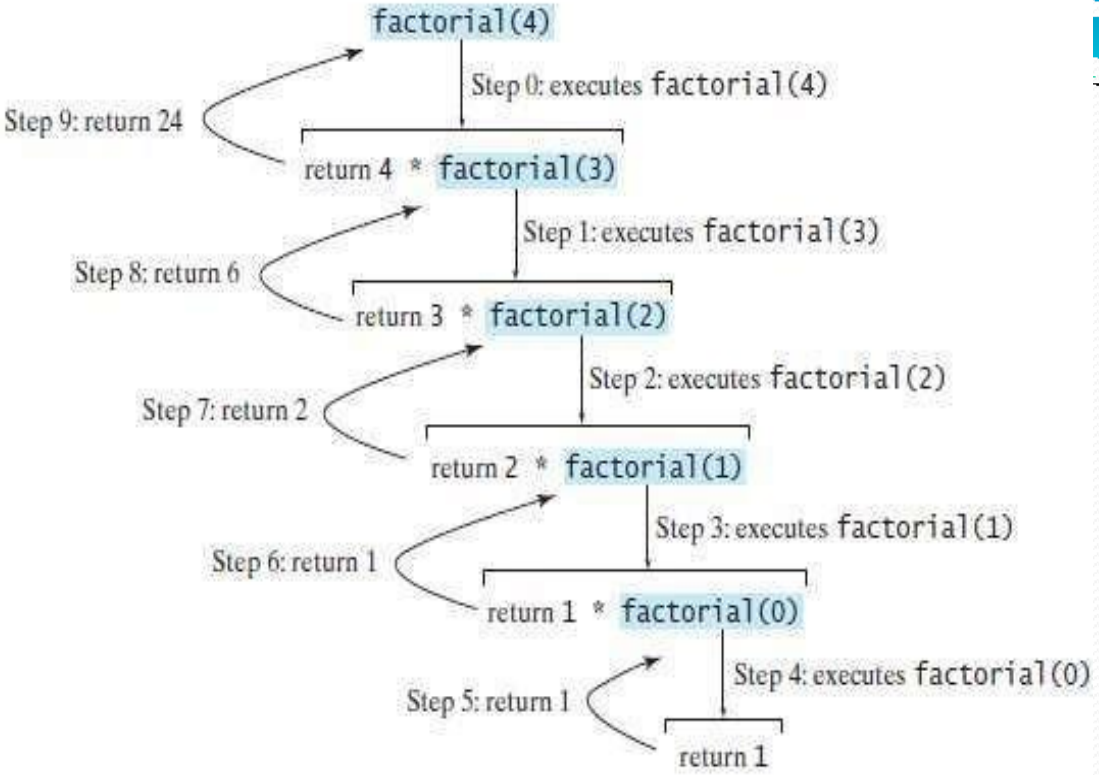
Factorial Program

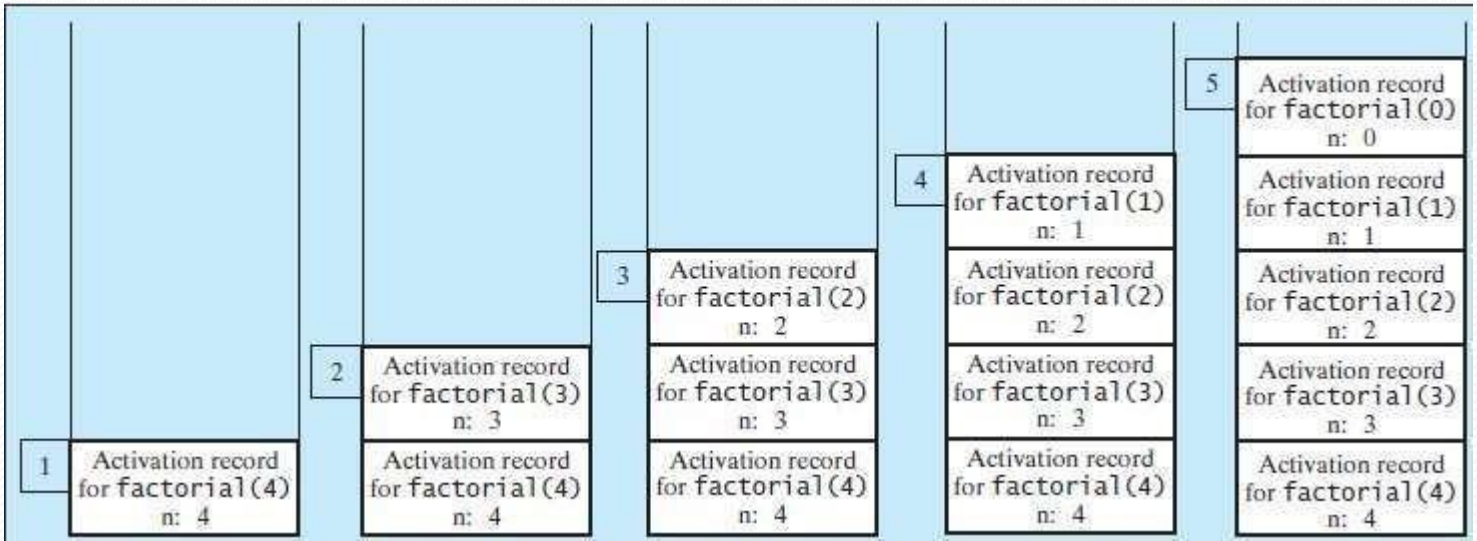
```
import java.util.Scanner;

public class ComputeFactorial {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a nonnegative integer: ");
    int n = input.nextInt();

    // Display factorial
    System.out.println("Factorial of " + n + " is " + factorial(n));
  }

  /** Return the factorial for the specified number */
  public static long factorial(int n) {
    if (n == 0) // Base case
      return 1;
    else
      return n * factorial(n - 1); // Recursive call
  }
}
```

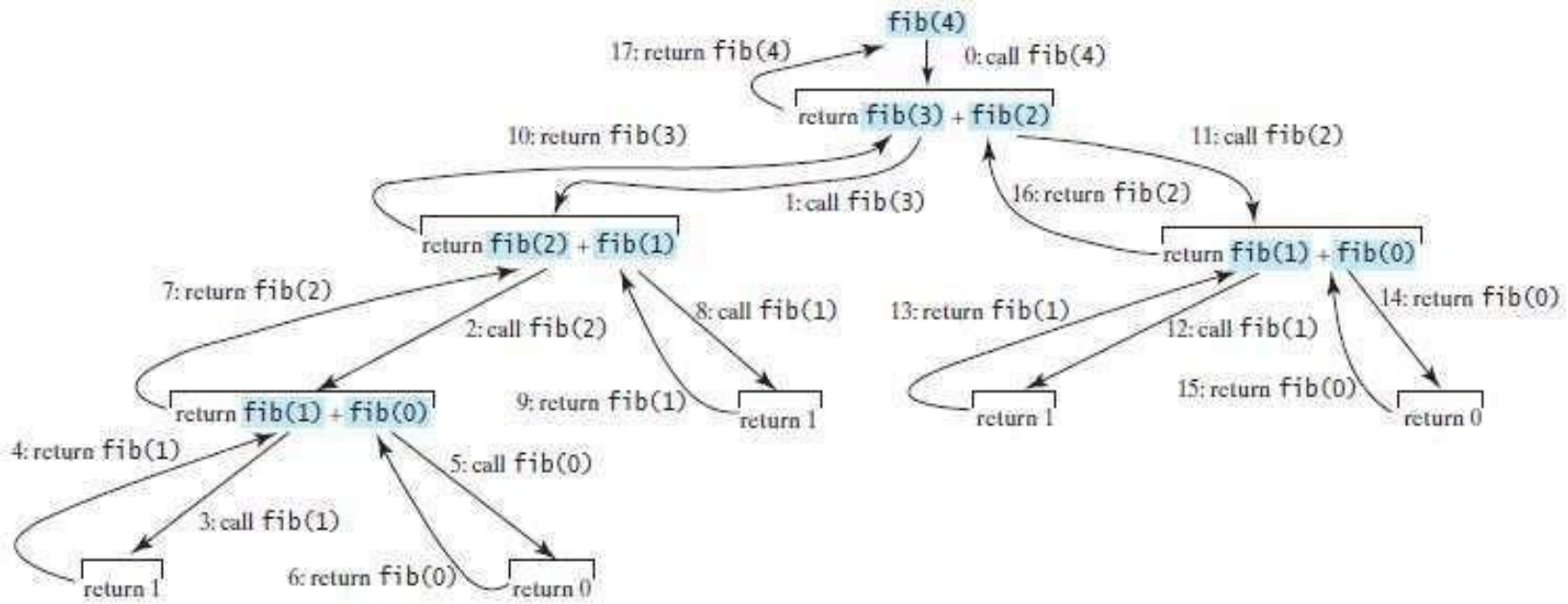

factorial(4)
Step 0: executes factorial(4)
Step 9: return 24
return 4 * factorial(3)
Step 1: executes factorial(3)
Step 8: return 6
return 3 * factorial(2)
Step 2: executes factorial(2)
Step 7: return 2
return 2 * factorial(1)
Step 3: executes factorial(1)
Step 6: return 1
return 1 * factorial(0)
Step 4: executes factorial(0)
Step 5: return 1
return 1


1
Activation record for factorial(4) n: 4
2
Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
3
Activation record for factorial(2) n: 2
Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
4
Activation record for factorial(1) n: 1
Activation record for factorial(2) n: 2
Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4
5
Activation record for factorial(0) n: 0
Activation record for factorial(1) n: 1
Activation record for factorial(2) n: 2
Activation record for factorial(3) n: 3
Activation record for factorial(4) n: 4

Fibonacci Series

- In some cases, recursion enables you to create an intuitive, straightforward, simple solution to a problem.

The series:	0	1	1	2	3	5	8	13	21	34	55	89	...
indexes:	0	1	2	3	4	5	6	7	8	9	10	11	



Problem Solving Using Recursion

- If you think recursively, you can solve many problems using recursion.
- Factorial and Fibonacci series are two classic recursion examples. All recursive methods have the following characteristics:
  - The method is implemented using an **if-else** or a **switch** statement that leads to different cases.
  - One or more base cases (the simplest case) are used to stop recursion.
  - Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Problem Solving Using Recursion

- Consider the palindrome problem.
- For example, “mom” and “dad” are palindromes, but “uncle” and “aunt” are not. The problem of checking whether a string is a palindrome can be divided into two sub problems:
  - Check whether the first character and the last character of the string are equal.
  - Ignore the two end characters and check whether the rest of the substring is a palindrome.

```java
public class RecursivePalindromeUsingSubstring {
  public static boolean isPalindrome(String s) {
    if (s.length() <= 1) // Base case
      return true;
    else if (s.charAt(0) != s.charAt(s.length() - 1)) // Base case
      return false;
    else
      return isPalindrome(s.substring(1, s.length() - 1));
  }

  public static void main(String[] args) {
    System.out.println("Is moon a palindrome? "
      + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? "
      + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a"));
    System.out.println("Is aba a palindrome? " +
      isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
  }
}
```

```
Is moon a palindrome? false
Is noon a palindrome? true
Is a a palindrome? true
Is aba a palindrome? true
Is ab a palindrome? false
```

Recursive Helper Methods

- Sometimes you can find a solution to the original problem by defining a recursive function to a problem similar to the original problem. This new method is called a recursive helper method. The original problem can be solved by invoking the recursive helper method.
- The recursive isPalindrome method is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, you can use the low and high indices to indicate the range of the substring.

```
public class RecursivePalindrome {
  public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length() - 1);
  }

  private static boolean isPalindrome(String s, int low, int high) {
    if (high <= low) // Base case
      return true;
    else if (s.charAt(low) != s.charAt(high)) // Base case
      return false;
    else
      return isPalindrome(s, low + 1, high - 1);
  }

  public static void main(String[] args) {
    System.out.println("Is moon a palindrome? "
      + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? "
      + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a"));
    System.out.println("Is aba a palindrome? " + isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
  }
}
```

Recursion vs. Iteration

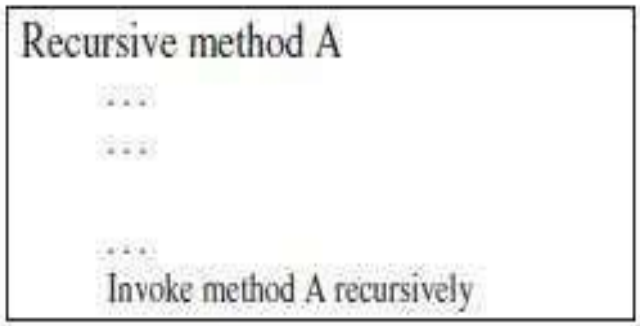
- Recursion is an alternative form of program control. It is essentially repetition without a loop.
- The repetition of the loop body is controlled by the loop control structure.
- In recursion, the method itself is called repeatedly. A selection statement must be used to control whether to call the method recursively or not.
- Recursion bears substantial overhead. Each time the program calls a method, the system must allocate memory for all of the method's local variables and parameters. This can consume considerable memory and requires extra time to manage the memory.

Recursion vs. Iteration

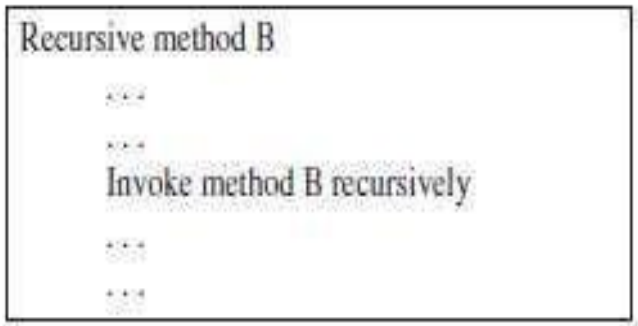
- Any problem that can be solved recursively can be solved nonrecursively with iterations. **Recursion has some negative aspects: it uses up too much time and too much memory.**
- Examples are the directory-size problem, the Tower of Hanoi problem, and the fractal problem, which are rather difficult to solve without using recursion.
- The decision whether to use recursion or iteration should be based on the nature of, and your understanding of, the problem you are trying to solve.

Tail Recursion

- A tail recursive method is efficient for reducing stack size.
- A recursive method is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.
- For example, the recursive isPalindrome method **is tail recursive** because there are no pending operations after recursively invoking isPalindrome.



(a) Tail recursion



(b) Nontail recursion

Tail Recursion

- The recursive factorial method **is not tail recursive**, because there is a pending operation, namely multiplication, to be performed on return from each recursive call.
- Tail recursion is desirable, because the method ends when the last recursive call ends, there is no need to store the intermediate calls in the stack. Compilers can optimize tail recursion to reduce stack size.

Factorial Program using Tail Recursion

```
public class ComputeFactorialTailRecursion {
  /** Return the factorial for a specified number */
  public static long factorial(int n) {
    return factorial(n, 1); // Call auxiliary method
  }

  /** Auxiliary tail-recursive method for factorial */
  private static long factorial(int n, int result) {
    if (n == 0)
      return result;
    else
      return factorial(n - 1, n * result); // Recursive call
  }
}
```

Content

- Generics
- Defining Generic Classes and Interfaces
- Generic Methods
- Raw Types and Backward Compatibility
- Wildcard Generic Types
- Erasure and Restrictions on Generics

Generics

- Generics enable you to detect errors at compile time rather than at runtime.
- Generics let you parameterize types. With this capability, you can define a class or a method with generic types that the compiler can replace with concrete types.
- For example, Java defines a generic ArrayList class for storing the elements of a generic type. From this generic class, you can create an ArrayList object for holding strings and an ArrayList object for holding numbers. Here, strings and numbers are concrete types that replace the generic type.

Generics

- <T> or <E> represents a generic type, which can be replaced later with an actual **concrete type**. Replacing a generic type is called a **generic instantiation.** Here, E or T is used to denote a formal generic type.

```
package java.lang;

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

```
ArrayList<AConcreteType> list = new ArrayList<>();

ArrayList<String> cities = new ArrayList<String>();

ArrayList<Integer> list = new ArrayList<>();
```

Generics

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

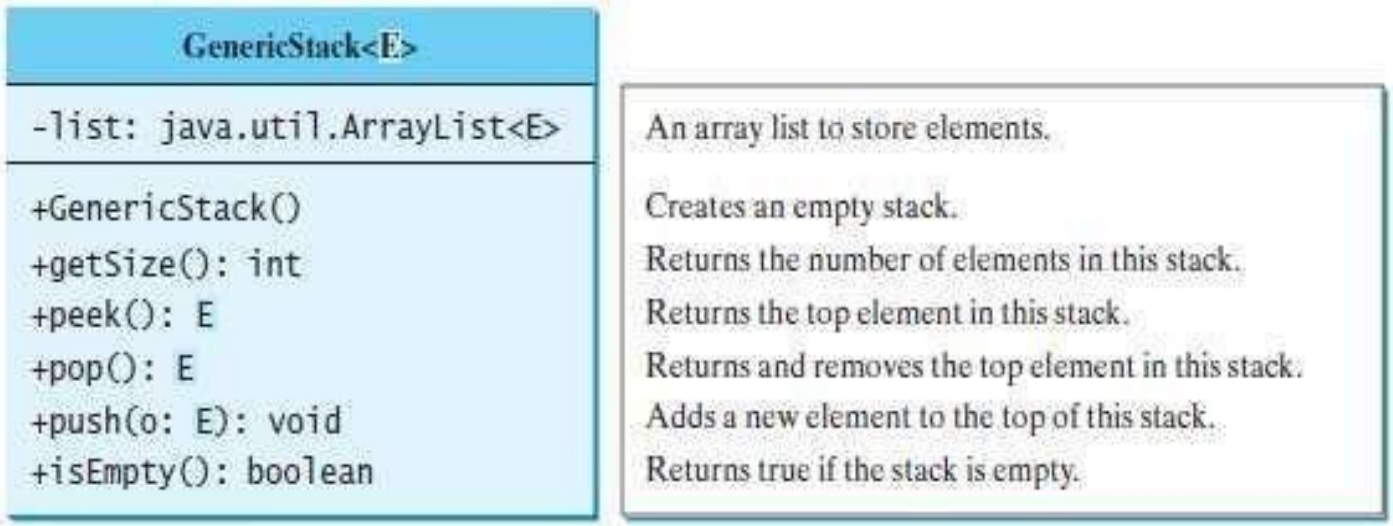
- Fig (a) : c is a reference variable whose type is Comparable and invokes the compareTo method to compare a Date object with a string. It will give run time error.
- Fig (b) : This code generates a compile error, because the argument passed to the compareTo method must be of the **Date type.**

Generics

- Type Parameters
- Naming conventions are important to learn in generics. Commonly used parameters are,
- T – Type
- E – Element
- K – Key
- N – Number
- V – Value

Defining Generic Classes and Interfaces

- A generic type can be defined for a class or interface. A concrete type must be specified when using the class to create an object or using the class or interface to declare a reference variable.



```java
public class GenericStack<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<>();

  public int getSize() {
    return list.size();
  }

  public E peek() {
    return list.get(getSize() - 1);
  }

  public void push(E o) {
    list.add(o);
  }

  public E pop() {
    E o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  @Override
  public String toString() {
    return "stack: " + list.toString();
  }
}
```

```java
GenericStack<String> stack1 = new GenericStack<>();
stack1.push("London");
stack1.push("Paris");
stack1.push("Berlin");
```

```java
GenericStack<Integer> stack2 = new GenericStack<>();
stack2.push(1); // autoboxing 1 to new Integer(1)
stack2.push(2);
stack2.push(3);
```

Generic Classes Example

```
class gen<T> //brackets indicates the class is of generic type // class Test<T, U>
{
T obj; //an object of type T is created
gen(T obj) {
this.obj =obj;
}
public void print() {
System.out.println(obj);
} }
class Demogen
{
public static void main(String args[]) {
gen<Integer> iob=new gen<Integer>(100);
iob.print();

gen<String> sob=new gen<String>("Hello");
sob.print();
} }
```

Output: 100
Hello

Generic Methods

- A generic type can be defined for a static method.
- Here, it defines a generic method **print** to print an array of objects.
- It passes an array of integer objects to invoke the generic print method and invokes print with an array of strings.

```
public class GenericMethodDemo {
  public static void main(String[] args ) {
    Integer[] integers = {1, 2, 3, 4, 5};
    String[] strings = {"London", "Paris", "New York", "Austin"};

    GenericMethodDemo.<Integer>print(integers);
    GenericMethodDemo.<String>print(strings);
  }

  public static <E> void print(E[] list) {
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println();
  }
}
```

Generic Methods

- To declare a generic method, you place the generic type <E> immediately after the keyword static in the method header.
- For example, public static <E> void print(E[] list)
- To invoke a generic method, prefix the method name with the actual type in angle brackets. If actual type is not explicitly specified, then compiler automatically discovers the actual type.

Generic Methods

```
class Test
{
 // A Generic method example
static void genericDisplay (T element) {
System.out.println(element.getClass().getName() +  " = " + element);
 }

 public static void main(String[] args)
 {
genericDisplay(11);

genericDisplay("Hello");

genericDisplay(1.0);
 }
}
```

Output :

```
java.lang.Integer = 11
java.lang.String = Hello
java.lang.Double = 1.0
```

Raw Types and Backward Compatibility

- A generic class or interface used without specifying a concrete type, called a raw type, enables backward compatibility with earlier versions of Java.
- For Ex. GenericStack stack = new GenericStack(); // **raw type**
- Which is **equivalent to**

GenericStack<Object> stack = new GenericStack<Object>();

- A generic class such as GenericStack and ArrayList used without a type parameter is called a **raw type**.
- Generic type has been used in java.lang.Comparable since JDK 1.5, but a lot of code still uses the raw type Comparable.

Raw Types and Backward Compatibility

```
public class Max {
  /** Return the maximum of two objects */
  public static Comparable max(Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

- Comparable o1 and Comparable o2 are raw type declarations. Be careful: *raw types are unsafe*.
- For example, you might invoke the max method using Max.max("**Welcome**", **23**);
- This would cause a runtime error, because you cannot compare a string with an integer object.

Raw Types and Backward Compatibility

- A betterway towrite the max method is touse a generictype.

  MaxUsingGenericType.max("**Welcome**", **23**);

- Here, compileerrorwill be displayed, because the twoarguments of the max method in MaxUsingGenericType must have the same type.

```
public class MaxUsingGenericType {
  /** Return the maximum of two objects */
  public static <E extends Comparable<E>> E max(E o1, E o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

Bounded Generics

```
public class BoundedTypeDemo {
  public static void main(String[] args ) {
    Rectangle rectangle = new Rectangle(2, 2);
    Circle circle = new Circle(2);

    System.out.println("Same area? " +
      equalArea(rectangle, circle));
  }

  public static <E extends GeometricObject> boolean equalArea(
      E object1, E object2) {
    return object1.getArea() == object2.getArea();
  }
}
```

- A generic type can be specified as a subtype of another type. Such a generic type is called **bounded**.

Bounded Generics

- Bounded type is used to limit or restrict the type of object passed to the parameterized type.

Ex: if we want a generic class to work only with numbers

Syntax: <T extends SuperClass> i.e. <T extends Number>

```
class gen<T extends Number> {
T obj;
gen(T obj) {
this.obj =obj;
}
public void print() {
System.out.println(obj);
} }
class Demogen {
public static void main(String args[]) {
gen<Integer> iob=new gen<Integer>(100);
iob.print();
//gen<String> sob=new gen<String>("Hello");
//sob.print();  } }
```

Wildcard Generic Types

- You can use unbounded wildcards, bounded wildcards, or lower-bound wildcards to specify a range for a generic type.
- To circumvent this problem, use wildcard generic types. A wildcard generic type has **3** forms:
- ? unbounded wildcard
- ? extends T, bounded wildcard
- ? super T, lower-bound wildcard
- Solution:

```
public static double max(GenericStack<? extends Number> stack) {
```

```
public class WildCardNeedDemo {
  public static void main(String[] args ) {
    GenericStack<Integer> intStack = new GenericStack<>();
    intStack.push(1); // 1 is autoboxed into new Integer(1)
    intStack.push(2);
    intStack.push(-2);

    System.out.print("The max number is " + max(intStack));
  }

  /** Find the maximum in a stack of numbers */
  public static double max(GenericStack<Number> stack) {
    double max = stack.pop().doubleValue(); // Initialize max

    while (!stack.isEmpty()) {
      double value = stack.pop().doubleValue();
      if (value > max)
        max = value;
    }

    return max;
  }
}
```

Error

Solution of Previous Example

```
public class AnyWildCardDemo {
  public static void main(String[] args ) {
    GenericStack<Integer> intStack = new GenericStack<>();
    intStack.push(1); // 1 is autoboxed into new Integer(1)
    intStack.push(2);
    intStack.push(-2);

    print(intStack);
  }

  /** Prints objects and empties the stack */
  public static void print(GenericStack<?> stack) {
    while (!stack.isEmpty()) {
      System.out.print(stack.pop() + " ");
    }
  }
}
```

Erasure and Restrictions on Generics

- The information on generics is used by the compiler but is not available at runtime. This is called type erasure.
- Generics are implemented using an approach called type erasure: The compiler uses the generic type information to compile the code, but erases it afterward. Thus, the generic information is not available at runtime. This approach enables the generic code to be backward compatible with the legacy code that uses raw types.

Erasure and Restrictions on Generics

```
ArrayList<String> list = new ArrayList<>();
list.add("Oklahoma");
String state = list.get(0);
```

(a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

(b)

- The generics are present at compile time. Once the compiler confirms that a generic type is used safely, it converts the generic type to a raw type.
- Here, compiler checks whether the following code in (a) uses generics correctly and then translates it into the equivalent code in (b) for runtime use.

Erasure and Restrictions on Generics

- When genericclasses, interfaces, and methods are compiled, the compiler replaces the generic type with the **Object type.**

```
public static <E> void print(E[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(a)

```
public static void print(Object[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(b)

- If a generictype is bounded, the compiler replaces it with the bounded type.

```
public static <E extends GeometricObject>
    boolean equalArea(
      E object1,
      E object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(a)

```
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(b)

Erasure and Restrictions on Generics

- For ex, ArrayList<String> and ArrayList<Integer> are two types at compile time, only one **ArrayList** class is loaded into the JVM at runtime.
- Because generic types are erased at runtime, there are certain restrictions on how generic types can be used.
  Restrictions:

1. **Restriction 1: Cannot Use new E()**

```
E object = new E();  // Not allowed
```

- Because new E() is executed at runtime, but the generic type E is not available at runtime.

Restrictions on Generics

2. **Restriction 2: Cannot Use new E[]**

```
E[] elements = new E[capacity];   // Not Allowed
```

3. **Restriction 3: A Generic Type Parameter of a Class Is Not Allowed in a Static Context**

- The static variables and methods of a generic class are shared by all its instances. Therefore, it is illegal to refer to a generic type parameter for a class in a static method, field, or initializer.

4. **Restriction 4: Exception Classes Cannot Be Generic**

End of chapter 9