Program 1 : Write a program to demonstrate the concept of boxing and unboxing

Algorithm :

1) Start
2) Declare and initialize a variable ‘i’ of datatype ‘int’
3) Declare an instance of wrapper class ‘Integer’ identified by ‘I’ and initialize with ‘i’
   /* This demonstrates the concept of “Boxing” */
4) Declare and initialize an instance of a wrapper class ‘Character’ identified by ‘CH’
5) Declare a variable of datatype ‘char’ and initialize it with ‘CH’
   /* This demonstrates the concept of “Unboxing” */
6) End

Program Code:

```
class BoxingAndUnboxing{
@SuppressWarnings("removal")
public static void main(String[] args){
 // Boxing
int i = 100;
Integer I = new Integer(i);
System.out.println("int : " + i);
System.out.println("[Boxing] int -> Integer : " + I);
// Unboxing
Character CH = new Character('x');
char ch = CH;
System.out.println("Character : " + CH);
System.out.println("[Unboxing] Character -> char : " + ch);
}
}
```

Output:

```
sh-5.1$ java BoxingAndUnboxing.java
int : 100
[Boxing] int -> Integer : 100
Character : x
[Unboxing] Character -> char : x
```

Discussion:

Here, the concept of Boxing and Unboxing has been explained and demonstrated.

Program 2 : To check if a number is prime or not, by taking the number as input from the keyboard

Algorithm :

1) Start
2) Declare int ‘num’ and initialize with user input
3) If (num < 1) then ;
   a) Print “input a positive integer” and repeat from step-2;
4) For (int i = 2; i*i <= num; i++) then;
   a) If (num % i == 0) then;
      i) Print “ $num is prime”
      ii) Exit Program
5) Print “$num is not prime”
6) End

Program Code:

```
import java.util.Scanner;
class CheckPrime{
public static void main(String[] args){
Scanner in = new Scanner(System.in);
int num;
boolean isPrimeFlag = true;
while(true){
System.out.print("Enter a positive integer : ");
num = in.nextInt();
if (num < 1) { System.out.println("Input a positive integer!"); continue; }
in.close();
break; }
for (int i = 2; i*i <= num; i++) { if(num % i == 0){ isPrimeFlag = !isPrimeFlag; break; } }
System.out.println(num + ( isPrimeFlag ? " is prime!" : " is not prime!"));
}
}
```

Output:

1) Set-1:
```
sh-5.1$ java CheckPrime.java
Enter a positive integer : -1
Input a positive integer!
Enter a positive integer : 9
9 is not prime!
```
2) Set-2:
```
sh-5.1$ java CheckPrime.java
Enter a positive integer : 17
17 is prime!
```

Discussion:

Time Complexity : O(sqrt n); Space Complexity: O(1);

Program 3 : To convert a decimal to binary number

Algorithm:

1) Start
2) Declare int 'num' and initialize with an integer [(2^{31}-1)~(-2^{31})] by user
3) int ptr := 2^{30}
4) While (ptr != 0) do;
    a) If ((num & ptr) == 0) then;
        i) Print 0;
    b) else
        i) Print 1;
    c) ptr = ptr / 2;
5) End

Program Code:

```
import java.util.Scanner;
class DecimalToBinary{
public static void main(String[] args){
Scanner in = new Scanner(System.in);
int num,  ptr = 1 << 30;
System.out.print("Enter the decimal integer : ");
num = in.nextInt();
in.close();
System.out.print("Binary representation -> ");
while ((ptr/=2) != 0){ System.out.print((num & ptr) != 0 ? "1" : "0"); }
System.out.println("");
}
}
```

Output:

1) Set-1:
    sh-5.1$ java DecimalToBinary.java
    Enter the decimal integer : 10
    Binary representation -> 000000000000000000000000001010

2) Set-2:
    sh-5.1$ java DecimalToBinary.java
    Enter the decimal integer : -100
    Binary representation -> 111111111111111111111110011100

Discussion:

Here, the binary representation of any integer [between (2^{31}-1) and (-2^{31})] can be printed using the bit manipulative techniques of programming.
Time Complexity: O(1)
Space Complexity: O(1)

Program 4 : To learn use of a single dimensional array by defining the array dynamically.

Algorithm:

1) Start
2) Declare int ‘size’ and initialize with user input
3) Allocate an array of ‘int’ of size ‘size’ on runtime
4) Input elements in every index of the array
5) Print array elements
6) End

Program Code:

```
import java.util.Scanner;
class DynamicMemoryAlloc{
public static void main(String[] args){
Scanner in = new Scanner(System.in);
System.out.print("Enter the size of array : ");
int size = in.nextInt();
in.close();
if ( size < 0 ) { System.out.println("Invalid size for memory allocation!"); return; }
int[] array = new int[size];
for (int i = 0; i < size; i++){
System.out.print("Enter elem-" + (i + 1) + " : ");
array[i] = in.nextInt();
}
System.out.print("Array Elements -> ");
for (int i = 0; i < size; i++){
System.out.print(" " + array[i]);
} System.out.println("");
}
}
```

Output:

1) Set-1:
   ```
   sh-5.1$ java DynamicMemoryAlloc.java
   Enter the size of array : 2
   Enter elem-1 : -10
   Enter elem-2 : 10
   Array Elements ->  -10 10
   ```
2) Set-2:
   ```
   sh-5.1$ java DynamicMemoryAlloc.java
   Enter the size of array : -10
   Invalid size for memory allocation!
   ```

Discussion:

Here, dynamic memory allocation at runtime has been demonstrated.

Program 5 : Find the factorial of a given number

Algorithm:

1) Start
2) Int num := (user input), factorial := 1;
3) If (num < 1) then;
    a) Repeat from step-2
4) While (num != 1) do;
    a) factorial = factorial * num;
    b) num = num - 1;
5) If (factorial != 0) then;
    a) Print factorial
6) End

Program Code:

```
import java.util.Scanner;
class Factorial{
public static void main(String[] args){
Scanner in = new Scanner(System.in);
int num, factorial = 1;
System.out.print("Enter a positive integer : ");
num = in.nextInt();
in.close();
if (num < 1){ System.out.println("Input beyond range!!"); return; }
factorial = num;
while(num-- != 1){ factorial *= num; }
if (factorial > 0) { System.out.println("Factorial : " + factorial); }
else { System.out.println("Precision lost! Factorial input range -> 1~16"); }
}
}
```

Output:

1) Set-1:
```
sh-5.1$ java Factorial.java
Enter a positive integer : -1
Input beyond range!!
```
2) Set-2:
```
sh-5.1$ java Factorial.java
Enter a positive integer : 10
Factorial : 3628800
```

Discussion:

Here, the above program returns the factorial of any integer given as user input.
Time Complexity: O(n);
Space Complexity: O(1);

Program 6 : Write a program to show that during function overloading, if no matching argument is found, then java will apply automatic type conversions(from lower to higher data type)

Algorithm:

1) Start
2) Function func(<lower_datatype_1> x_1) :
    a) Convert x_1 to higher any data type and print
3) End func
4) Function func(<lower_datatype_2> x_2) :
    a) Convert x_2 to higher any data type and print
5) End func
6) Function main :
    a) Declare x_3 and x_4 with lower_datatype_3 and lower_datatype_4 respectively.
    b) Call func(x_3)
    c) Call func(x_4)
7) End main
8) End

Program Code:

```
import java.util.Scanner;
class OverloadedFunctionsWithTypeConversion{
public static char func(short x){
System.out.println( x + " --(short to char)--> " + (char)(x));
return (char)(x);
}
public static void func(int x){
System.out.println( x + " --(int to double)--> " + (double)(x));
}
public static void main(String[] args){
Scanner in = new Scanner(System.in);
byte num_b;
System.out.print("Enter a byte code -> ");
num_b = in.nextByte();
in.close();
char ch = func(num_b);
func(ch);
}
}
```

Output:

1) Set-1:
   sh-5.1$ java OverloadedFunctionsWithTypeConversion.java
   Enter a byte code -> 100
   100 --(short to char)--> d
   100 --(int to double)--> 100.0

2) Set-2:
   sh-5.1$ java OverloadedFunctionsWithTypeConversion.java
   Enter a byte code -> 69
   69 --(short to char)--> E
   69 --(int to double)--> 69.0

Discussion:

Here, in the above code any byte code has been taken as user input and then the concept of automatic type conversion while function overloading has been demonstrated.

Program 7 : Write a program to demonstrate multi thread communication by implementing synchronization among threads (Hint: you can implement a simple producer and consumer problem).

Algorithm:

1) Define a shared `Message` object with a `content` variable and an `available` flag.
2) Implement the `put(message)` method in the `Message` class:
    a) Acquire the lock on the `Message` object.
    b) While `available` is `true`, wait for the consumer to consume the message by calling `wait()`.
    c) Set the `content` variable to the given `message`.
    d) Set `available` to `true`.
    e) Notify the consumer thread by calling `notify()`.
    f) Release the lock on the `Message` object.
3) Implement the `take()` method in the `Message` class:
    a) Acquire the lock on the `Message` object.
    b) While `available` is `false`, wait for the producer to put a message by calling `wait()`.
    c) Get the value of the `content` variable.
    d) Set `available` to `false`.
    e) Notify the producer thread by calling `notify()`.
    f) Release the lock on the `Message` object.
    g) Return the message.
4) Define a `Producer` class that implements the `Runnable` interface:
    a) Declare a `Message` object.
    b) Implement the `run()` method:
        i) Create an array of messages.
        ii) Iterate over the messages:
            (1) Call the `put(message)` method of the `Message` object to put the message.
            (2) Print the produced message.
            (3) Simulate some processing time using `Thread.sleep()`.
5) Define a `Consumer` class that implements the `Runnable` interface:
    a) Declare a `Message` object.
    b) Implement the `run()` method:
        i) Iterate a fixed number of times (e.g., 3):
            (1) Call the `take()` method of the `Message` object to take the message.
            (2) Print the consumed message.
            (3) Simulate some processing time using `Thread.sleep()`.

6) In the `main` method:
    a) Create an instance of the `Message` class.
    b) Create instances of the `Producer` and `Consumer` classes, passing the `Message` object.
    c) Create threads for the producer and consumer using the `Thread` class, passing the respective `Runnable` objects.
    d) Start both threads using the `start()` method.

Program Code:

```
public class MultithreadCommunicationDemo {
        public static void main(String[] args) {
        Message message = new Message();
        Thread producerThread = new Thread(new Producer(message));
        Thread consumerThread = new Thread(new Consumer(message));
        producerThread.start();
        consumerThread.start();
}

// Shared message object between producer and consumer threads
static class Message {
        private String content;
        private boolean available = false;
        public synchronized void put(String message) {
        // Wait until the message is consumed by the consumer
        while (available) {
        try { wait();
        } catch (InterruptedException e) { e.printStackTrace(); }
        }
        // Put the message
        content = message;
        available = true;
        // Notify the consumer thread that the message is available
        notify();
        }
        public synchronized String take() {
        // Wait until there is a message available from the producer
        while (!available) {
                try { wait();
                } catch (InterruptedException e) {e.printStackTrace();}
        }
        // Take the message
        String message = content;
        available = false;
```

```
        // Notify the producer thread that the message has been consumed
        notify();
        return message;
        }
}

        // Producer thread
static class Producer implements Runnable {
        private final Message message;
        public Producer(Message message) { this.message = message; }
        @Override
        public void run() {
        String[] messages = {"Message 1", "Message 2", "Message 3"};
        for (String msg : messages) {
        message.put(msg);
        System.out.println("Produced: " + msg);

        try {
                Thread.sleep(1000); // Simulate some processing time
        } catch (InterruptedException e) { e.printStackTrace(); }
        }
        }
}

        // Consumer thread
static class Consumer implements Runnable {
        private final Message message;
        public Consumer(Message message) { this.message = message; }
        @Override
        public void run() {
        for (int i = 0; i < 3; i++) {
        String receivedMessage = message.take();
        System.out.println("Consumed: " + receivedMessage);

        try {
                Thread.sleep(1000); // Simulate some processing time
        } catch (InterruptedException e) { e.printStackTrace(); }
        }
        }
        }
}
```

Output:

Produced: Message 1
Consumed: Message 1
Produced: Message 2
Consumed: Message 2
Produced: Message 3
Consumed: Message 3

Discussion:

The synchronization between the producer and consumer is achieved using the `synchronized` keyword in the `put()` and `take()` methods, ensuring that only one thread can access the shared `Message` object at a time. The `wait()` and `notify()` methods are used for signaling and waiting for the availability of messages, allowing proper coordination between the threads.