Introduction to Programming

Java – I/O, Operators, Control Flow, Arrays, Exceptions

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

January, 2021

Taking inputs (with Scanner class)

```
import java.io.*;
import java.util.Scanner;
class Input{
    public static void main(String args[]) throws IOException{
        int i;
        float f;
        Scanner sc = new Scanner(System.in);
        System.out.print("Age: ");
        i = sc.nextInt();
        if(i < 0)
            System.out.print("Invalid age!!!");
        else{
            System.out.print("Valid age. Now enter height: ");
            f = sc.nextFloat();
            System.out.print("Height = "+f);
        }
    }
}
```

Taking inputs (with DataInputStream class)

```
import java.io.*;
class Input{
    public static void main(String args[]) throws IOException{
        int i;
        double f;
        boolean c;
        DataInputStream dis = new DataInputStream(System.in);
        System.out.print("Age, height and code: ");
        i = dis.readInt();
        f = dis.readDouble();
        c = dis.readBoolean();
        System.out.print("Age = " + i + ", height = " + f +
        " feet" + " and code = " + c);
    }
}
```

Taking inputs (with DataInputStream class - alternatively)

```
import java.io.*;
class Input{
    public static void main(String args[]) throws IOException{
        int i;
        float f;
        char c;
        DataInputStream dis = new DataInputStream(System.in);
        System.out.print("Age, height and code: ");
        i = Integer.parseInt(dis.readLine());
        f = Float.valueOf(dis.readLine()).floatValue();
        c = (char) System.in.read();
        System.out.print("Age = " + i + ", height = " + f +
        " feet" + " and code = " + c);
    }
}
```

Showing outputs

```
System.out.println("String" + variables);
```

Showing outputs

```
System.out.println("String" + variables);
```

```
class Input{
    public static void main(String args[]){
        int i=10;
        // The following two statements are same
        System.out.println("Value of i = " + i);
        System.out.print("Value of i = " + i + "\n");
    }
}
```

Showing outputs

```
System.out.println("String" + variables);
```

```
class Input{
    public static void main(String args[]){
        int i=10;
        // The following two statements are same
        System.out.println("Value of i = " + i);
        System.out.print("Value of i = " + i + "\n");
    }
}
```

Note: Java being strongly typed the format specifiers are not required.

Arithmetic operators

Summation $(+)$

Subtraction $(-)$

Multiplication $(*)$

Division $(/)$

Modulo division $(\%)$

Arithmetic operators

Summation $(+)$

Subtraction $(-)$

Multiplication $(*)$

Division $(/)$

Modulo division $(\%)$

Note: Modulo division operator also work on floats.

Relational operators

Less than $(<)$

Less than equals to $(<=)$

Greater than $(>)$

Greater than equals to $(>=)$

Equals to $(==)$

Not equals to $(!=)$

Logical operators

Logical and (&&)

Logical or (||)

Logical not (!)

Assignment operators

Assignment $(=)$

Increment and decrement operators

Increment $(++)$

Decrement $(--)$

Conditional operators

Ternary (? :)

```
int i = 0;
int j = 1;
int result = (i > j) ? i : j;
```

Bitwise operators

Bitwise and $(\&)$

Bitwise or $(|)$

Bitwise not $(\sim)$

Bitwise xor $(\wedge)$

Left shift $(<<)$

Right shift $(>>)$

Right shift with zero padding $(>>>)$

Special operators

Dot (.)

```
class Input{
    public static void main(String args[]){
        Summation s = new Summation();
        s.Add();
    }
}
```

if-else

```
if(Condition){
    statement 1;
    statement 2;
}
else
    statement 3; // Execute if Condition fails
```

if-else

```
if(Condition){
    statement 1;
    statement 2;
}
else
    statement 3; // Execute if Condition fails
```

Note: The 'Condition' should be in `Boolean` form.

for loop

```
for(Initialization;Condition;Increment/decrement){
    statement 1;
    statement 2;
}
```

for loop

```
for(Initialization;Condition;Increment/decrement){
    statement 1;
    statement 2;
}
```

Alternatively, we can write

```
for(Initialization;Condition;Increment/decrement)
    statement 3;
```

for loop

```
for(Initialization;Condition;Increment/decrement){
    statement 1;
    statement 2;
}
```

Alternatively, we can write

```
for(Initialization;Condition;Increment/decrement)
    statement 3;
```

Note: The 'Condition' should be in `Boolean` form.

while loop

```
while(Condition){
    statement 1;
    statement 2;
}
```

while loop

```
while(Condition){
    statement 1;
    statement 2;
}
```

Alternatively, we can write

```
while(Condition)
    statement 3;
```

while loop

```
while(Condition){
    statement 1;
    statement 2;
}
```

Alternatively, we can write

```
while(Condition)
    statement 3;
```

Note: The 'Condition' should be in `Boolean` form.

do-while loop

```
do{
    statement 1;
    statement 2;
}while(Condition)
```

do-while loop

```
do{
    statement 1;
    statement 2;
}while(Condition)
```

Note: The 'Condition' should be in `Boolean` form.

One dimensional array

Creating an array:

```
class Array{
    public static void main(String args[]){
        int semester[]; // Declaration
        semester = new int[6]; // Memory allocation
    }
}
```

One dimensional array

Creating an array:

```
class Array{
    public static void main(String args[]){
        int semester[]; // Declaration
        semester = new int[6]; // Memory allocation
    }
}
```

The declaration and memory allocation can be combined together as follows:

```
int semester[] = new int[6];
```

One dimensional array

Creating an array:

```
class Array{
    public static void main(String args[]){
        int semester[]; // Declaration
        semester = new int[6]; // Memory allocation
    }
}
```

The declaration and memory allocation can be combined together as follows:

```
int semester[] = new int[6];
```

Note: The declaration can also be like "`int[] semester;`"

One dimensional array

Initializing an array:

```
class Array{
    public static void main(String args[]){
        // Initialization while declaring
        int semester[] = {1,2,3,4,5,6};
    }
}
```

One dimensional array

Initializing an array:

```
class Array{
    public static void main(String args[]){
        // Initialization while declaring
        int semester[] = {1,2,3,4,5,6};
    }
}
```

Note: The length of the array can be returned with "`semester.length;`"

Two dimensional array

Creating an array:

```
class Array{
    public static void main(String args[]){
        // Declaration and allocation
        float result[][] = new float[6][30];
    }
}
```

Variable size array

Creating an array of variable sizes:

```
class Array{
    public static void main(String args[]){
        char result[][] = new char[5][]; // No. of rows
        result[0] = new char[10]; // Row 1 has 10 columns
        result[1] = new char[20]; // Row 2 has 20 columns
        result[2] = new char[30]; // Row 3 has 30 columns
        result[3] = new char[10]; // Row 4 has 10 columns
        result[4] = new char[50]; // Row 5 has 50 columns
    }
}
```

Strings

Creating a string:

```
class ABCD{
    public static void main(String args[]){
        String name; // Declaration
        name = new String("Y-"); // Initialization
    }
}
```

Strings

Creating a string:

```
class ABCD{
    public static void main(String args[]){
        String name; // Declaration
        name = new String("Y-"); // Initialization
    }
}
```

The declaration and initialization can be combined together as follows:

```
String name = new String("Y-");
```

Strings

Creating an array of strings:

```
class StrArray{
    public static void main(String args[]){
        // Declaration
        String name[] = new String[10];
    }
}
```

Strings

Creating an array of strings:

```
class StrArray{
    public static void main(String args[]){
        // Declaration
        String name[] = new String[10];
    }
}
```

Note: Java strings can be concatenated with the '+' operator.

Strings

Creating an array of strings:

```
class StrArray{
    public static void main(String args[]){
        // Declaration
        String name[] = new String[10];
    }
}
```

Note: Java strings can be concatenated with the '+' operator. E.g., "`String name = "Life" + "of" + "Pi";`" will result "LifeofPi".

Vectors

The `Vector` class implements a growable array of objects. However, its size can grow/shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Vectors

The `Vector` class implements a growable array of objects. However, its size can grow/shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Creating a vector:

```
class V{
    public static void main(String args[]){
        Vector obj1 = new Vector(); // Without size
        Vector obj2 = new Vector(10); // With size
    }
}
```

Vectors

The `Vector` class implements a growable array of objects. However, its size can grow/shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Creating a vector:

```
class V{
    public static void main(String args[]){
        Vector obj1 = new Vector(); // Without size
        Vector obj2 = new Vector(10); // With size
    }
}
```

Similar functionality can also be obtained with the `ArrayList` class.

Vectors

The `Vector` class implements a growable array of objects. However, its size can grow/shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Creating a vector:

```
class V{
    public static void main(String args[]){
        Vector obj1 = new Vector(); // Without size
        Vector obj2 = new Vector(10); // With size
    }
}
```

Similar functionality can also be obtained with the `ArrayList` class.

Note: While using vectors you should include "`import java.util.*`"

Differentiating the `Vector` class from `ArrayList`

`Vector`	`ArrayList`
1. `Vectors` are synchronized.	1. `ArrayList` is unsynchronized.
2. Any method that touches the `Vector`'s contents is thread safe.	2. `ArrayList` is not thread safe.
3. Data growth can be efficiently handled with `Vectors`.	3. Data growth is not efficient to handle with `ArrayList`.

Exception handling

```
try{
    statement 1;
    statement 2;
}
catch(Exception type 1){
    statement 3; // Executed for exception type 1
}
catch(Exception type 2){
    statement 4; // Executed for exception type 2
}
```

Exception handling

```
try{
    statement 1;
    statement 2;
}
catch(Exception type 1){
    statement 3; // Executed for exception type 1
}
catch(Exception type 2){
    statement 4; // Executed for exception type 2
}
```

Note: A `try` block should be matched with at least one `catch` or `finally` block.

Use of `finally` in exception handling

A `finally` block can also be matched with a `try` block to handle exceptions. Once entered, the `finally` block will always execute no matter if you return from it or an exception is thrown within the try block.

Use of `finally` in exception handling

A `finally` block can also be matched with a `try` block to handle exceptions. Once entered, the `finally` block will always execute no matter if you return from it or an exception is thrown within the try block.

Only if you call `exit()` before the `finally` is reached, it will skip the execution.

An example

```
import java.io.DataInputStream;
class Input{
    public static void main(String args[]){
        int i;
        DataInputStream in = new DataInputStream(System.in);
        System.out.println("Enter the value = ");
        try{
            i = Integer.parseInt(in.readLine());
        }
        catch(Exception e){ // 'e' can be renamed
            System.out.print(e);
        }
    }
}
```

`catch` exception types

- **IOException:** Failure in taking input or showing output
- **ArithmeticException:** Arithmetic errors (e.g., divide by zero)
- **ArrayIndexOutOfBoundsException:** Array index out of bounds
- **StringIndexOutOfBoundsException:** String index out of bounds
- **NegativeArraySizeException:** Negative array size error
- **NullPointerException:** Referencing a null object
- **NumberFormatException:** Failure of string to number conversion
- **OutOfMemoryException:** Not enough memory
- **SecurityException:** Applet causes security problems

... Find more in books.

Throwing exception for a single statement

```
import java.io.DataInputStream;
class Input{
    public static void main(String args[]){
        int i;
        DataInputStream in = new DataInputStream(System.in);
        System.out.println("Enter the value = ");
        // No try-catch required
        i = Integer.parseInt(in.readLine());
        if(i == 0){
            throw new IOException("Unable to operate");
        }
        System.out.print(i);
    }
}
```

Throwing exception for an entire method

```
import java.io.DataInputStream;
class Input{
    public static void main(String args[]) throws IOException{
        int i;
        DataInputStream in = new DataInputStream(System.in);
        System.out.println("Enter the value = ");
        // No try-catch required
        i = Integer.parseInt(in.readLine());
        System.out.print(i);
    }
}
```

Nested try-catch

```
try{
    statement 1;
    try{ // Nested try
        statement 2;
    }
    catch(Exception type 1){
        statement 3;
    }
}
catch(Exception type 2){
    try{
        statement 4;
    }
    catch(Exception type 3){
        statement 5;
        statement 6;
    }
}
```