# JAVA

Lecture VI – I/O

# I/O

- I/O system in Java:

# I/O

- Stream: an abstraction that either produces or consumes information
  - Linked with physical devices
  - Same I/O classes can be applied to different devices
  - Java's I/O classes are defined in the **java.io** package
- Two types of streams:
  - Byte streams: handling binary data. (Useful for files)
  - Character streams: handling characters. (more efficient)

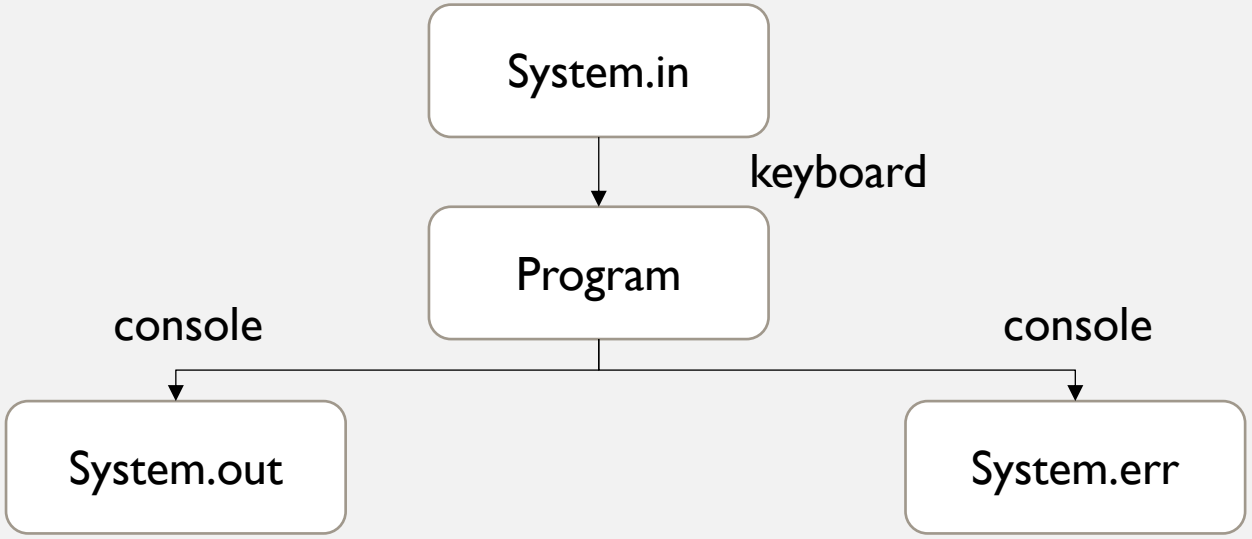
# BYTE AND CHARACTER STREAM

- Two classes of Byte Stream
  - InputStream:  defines characteristics to byte input streams.
  - OutputStream: describes the behaviour of byte output streams.

- Two classes of Character Stream
  - Reader: for input
  - Writer: for output

# PREDEFINED STREAMS

- Three predefined stream variables (all declared as **public**, **final**, and **static**):
  - System.in: the standard input stream (the keyboard).
  - System.out: the standard output stream (the console).
  - System.err: the standard error stream (the console).



# READING CONSOLE INPUT

- Both Byte Stream and Character Stream can be used to perform console input.
- We will learn Character-based Stream based on **BufferedReader**.
- Can we do

```
BufferedReader br = new BufferedReader(System.in);
```

# READING CONSOLE INPUT

- Both Byte Stream and Character Stream can be used to perform console input.
- We will learn Character-based Stream based on **BufferedReader**.
- Can we do

```
BufferedReader br = new BufferedReader(System.in);
```

- No, System.in is actually a byte stream.

# READING CONSOLE INPUT

- Both Byte Stream and Character Stream can be used to perform console input.
- We will learn Character-based Stream based on **BufferedReader**.
- Can we do

```
BufferedReader br = new BufferedReader(System.in);
```

- No, System.in is actually a byte stream.

```
InputStreamReader sr = new InputStreamReader(System.in);
```

- It convert a byte stream to a character stream (a subclass of Reader).

# READING CONSOLE INPUT

- Both Byte Stream and Character Stream can be used to perform console input.
- We will learn Character-based Stream based on **BufferedReader**.
- Can we do

```
BufferedReader br = new BufferedReader(System.in);
```

- No, System.in is actually a byte stream.

```
InputStreamReader sr = new InputStreamReader(System.in);
```

- It convert a byte stream to a character stream (a subclass of Reader).
- `BufferedReader br = new BufferedReader(sr);`
- Now br is a character stream linked to the console through System.in

# READING CHARACTERS

- To read a single character:
- `int read() throws IOException`
- Be careful there is a checked exception to handle here.
- read() reads a single character from the input stream.

```
BufferedReader br = new BufferedReader(
                    new InputStreamReader(System.in));
char c = (char)br.read();      // read() returns an integer
```

# READING STRINGS

- To read a string from the keyboard:
- `String readLine() throws IOException`
- It returns a String object
- readLine() reads a line of input from the input stream.

```
do{
   str = br.readLine();
   System.out.println(str);
}while(!str.equals("stop"));
```

# SCANNER

- Scanner is a class in java.util used for obtaining the input of the primitive types.
- It is much easier to use compared to BufferedReader, but slower.

```
Scanner sr = new Scanner(System.in);
```

- To read a String:

```
sr.nextLine();
```

- To read other primitive types:

```
nextInt(), nextFloat(), nextDouble(),nextShort(), …
```

- An example of Scanner

# FILE I/O USING CHARACTER STREAM

- FileReader: create a reader you can use to read the content of a file.

```
FileReader(String fileName) throws FileNotFoundException
```

- `fileName` is the name of a file (could be the path to this file).
- This class throws an exception if the file does not exist.
- We need to create an object of BufferedReader using the FileReader as the input stream.
- Also need to handle the potential exceptions.

# TRY-WITH-RESOURCE

- When FileReader is used to access a file, we need to close it at the end.

```
FileReader fr = null; BufferedReader br = null;
try{
      fr = new FileReader("1.txt");
      br= new BufferedReader(fr);
      br.readLine();
}catch(IOException e){

}finally{
      try{
            fr.close();
      }catch(IOException e){

      }
}
```

# TRY-WITH-RESOURCE

```
try(declare and create resource)
```

When the resource will be automatically released (closed)

```
try(BufferedReader br = new BufferedReader(new FileReader("1.txt"))){
      br.nextLine();
}catch(FileNotFoundException e){
      // do something
}
```

# READ LINES FROM THE FILE

```
public static void main(String[] args){
   try(BufferedReader br = new BufferedReader(new FileReader("2.txt"))){
          while(br.readLine() != null){//if it is not the end
            System.out.println(br.readLine());
          }
   }
   catch(IOException e){
          System.out.println("file not found");
   }
}
```

# SCANNER

```
public static void main(String[] args){
    try(Scanner sr = new Scanner(new FileReader(”1.txt"))){
        String str;
        while(sr.hasNextLine()){   // check if it is the end of the file
            str = sr.nextLine();
            System.out.println(str);
        }
    }
    catch(IOException e){
        e.printStackTrace();
    }
}
```

# FILEWRITER

- FileWriter: create a writer you can use to write to a file.

```
FileWriter(String fileName) throws IOException

FileWriter(String fileName, boolean append) throws IOException
```

- `fileName` is the name of a file.
- We append the output to the end of the file, if append is true
- Both throws an IOException on failure.

# WRITE INPUT TO A FILE

```
public static void main(String[] args){
   try(FileWriter fw = new FileWriter("1.txt")){
      str = "new line";
      fw.write(str);
   }
   catch(IOException e){
         e.printStackTrace();
   }
}
```

# INNER CLASSES

- Nested classes:
  - Non-static nested classes: Inner classes
  - Static nested classes
- Group classes together to make code more readable and maintainable
- Scope: bounded by its outer class, all variables and methods of its outer class.
- General Form:

```
class Outer{
      class Inner{
      }
}
```

# INNER CLASS

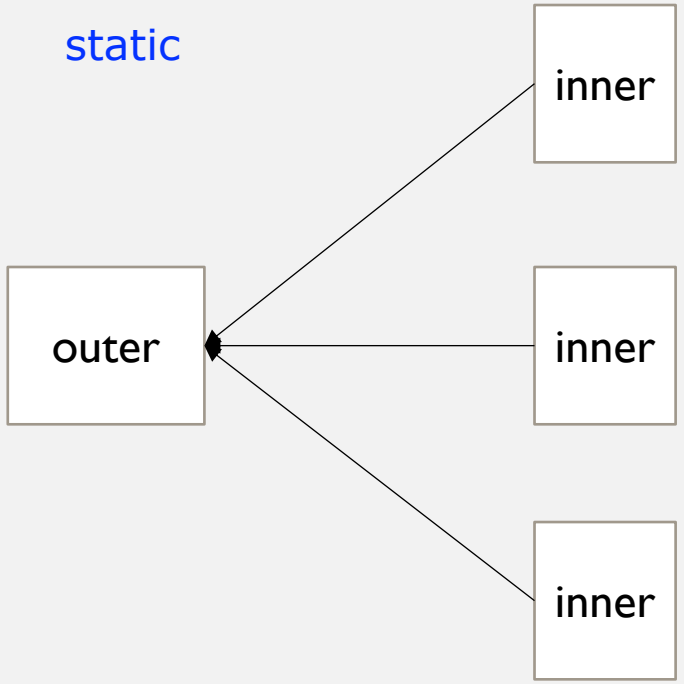
```
public class Outer{
	public int x = 10;
	public class Inner{
		public int y = 5;
		public void printX(){
			System.out.println(x);
		}
	}

	public int addition(){
		Inner inner = new Inner();
		return x + inner.y;
	}
}
```

# INNER CLASS

```
public class Outer{
	public int x = 10;
	public class Inner{
		public int y = 5;
		public void printX(){
			System.out.println(x);
		}
	}

	public int addition(){
		Inner inner = new Inner();
		return x + inner.y;
	}
}
```



# INNER CLASS

```
public class Outer{
	public int x = 10;
	public class Inner{
		public int y = 5;
	}
}

public class Test{
	public static void main(String[] args){
		Outer outer = new Outer();
		Outer.Inner inner = outer.new Inner();
		System.out.println(”(”outer.x + “,” + inner.y +”)”);
	}
}
```

# INNER CLASS

```
public class Outer{
	public int x = 10;
	private class Inner{
		public int y = 5;
	}
}

public class Test{
	public static void main(String[] args){
		Outer outer = new Outer();
		Outer.Inner inner = outer.new Inner();
		System.out.println(”(”outer.x + “,” + inner.y +”)”);
	}
}
```

# ACCESS PRIVATE VALUES

```
public class Outer{
	private int x = 10;
	public class Inner{
		public int y = 5;
		public int getValue(){
			return x;
		}
	}
}

public class Test{
	public static void main(String[] args){
		Outer outer = new Outer();
		Outer.Inner inner = outer.new Inner();
		System.out.println(inner.getValue());
	}
}
```

# METHOD-LOCAL INNER CLASSES

- In Java, an inner class can be defined within a method.
- Its scope is within the method, like local variables

```
public class LocalInner{
      public void aMethod{
            class Inner{
            }
            Inner inner = new Inner();
      }
}
```

# METHOD OVERLOAD

- Polymorphism: Redefine the way a class works by changing how it works or by changing the data.
  - Overload.
  - Override.
- Overload: two or more methods within the same class can share the same name, but with different parameter declarations.
- Restrictions:
  - The type and/or number of the parameters must differ.
  - The return types can be different, but not necessarily.

# EXAMPLE: OVERLOAD

```
class Overload{
    void ovlDemo(){
       System.out.println("No parameters");
    }

    void ovlDemo(int a){
       System.out.println("One parameter:" + a);
    }

    int ovlDemo(int a, int b){
       System.out.println("Two parameters: " + a + " " + b);
       return a + b;
    }
}
```

# OVERLOAD CONSTRUCTOR

- Constructors can also be overloaded.

```
class MyClass{
	int x;
	MyClass(){
		System.out.println(x);
	}
	MyClass(int i){
		x = i;
		System.out.println(x);
	}
	MyClass(int i, int j){
		x = i + j;
		System.out.println(x);
	}
}
```

# OVERLOAD CONSTRUCTORS

- Overloading constructors allows to use one object to initialise another

```
class Summation{
	int sum;
	Summation(int num){
		sum = 0;
		for(int i = 1; i <= num; i++){
			sum += i;
		}
	}
	Summation(Summation ob){
		sum = ob.sum;
	}
}
```

# FOR EACH

read-only for loop

- `int[] xs = {1, 2, 3, -5};`
- Examine each element in an array xs

```
for(int i =0; i < xs.length; i++)
```

- "for-each" style loop, for a collection of objects such as array

```
for(type itr-var : collection) statement-block
```

```
for(int x : xs){
        …
}
```

- Question: which one is better? For or foreach?

# EXAMPLE

```
int[] nums = {1, 11, 22, 3, 4, 66, 77, 2, 100, 212};
      for(int num : nums){
            num = 0;
      }
      for(int i = 0; i < nums.length; i++){
            System.out.println(nums[i]);
      }
```

- What is the result?

# EXAMPLE

- Given a list of numbers, return the first 4 numbers that is greater than 10.

```
int[] nums = {1, 11, 22, 3, 4, 66, 77, 2, 100, 212};
int[] xs = new int[4];
int index = 0;
for(int num: nums){
      if(num > 10){
            xs[index++] = num;
      }
}
```

- What is the problem?

# EXAMPLE

- Given a list of numbers, return the first 4 numbers that is greater than 10.

```
int[] nums = {1, 11, 22, 3, 4, 66, 77, 2, 100, 212};
int[] xs = new int[4];
int index = 0;
for(int num: nums){
      if(num > 10){
            xs[index++] = num;
      }
}
//We want the loop stops immediately when we find 4 numbers
```

- What is the problem?

# BREAK

- Break is used to terminate current loop.

```
int[] nums = {1, 11, 22, 3, 4, 66, 77, 2, 100, 212};
int[] xs = new int[4];
int index = 0;
for(int num: nums){
      if(num > 10){
            xs[index++] = num;
            if(index >= 4)
                  break;
      }
}
```

# ANOTHER EXAMPLE

- Print "Invalid" if the summation of a number from xs and another number from ys is greater than 40.

```
int[] xs = {1, 11, 22, 3, 4, 66, 77, 2, 100, 212};
int[] ys = {3, 30, 50, 23, 99};
for(int x: xs){
        for(int y: ys){
                if(x + y > 40){
                        System.out.println("Invalid");
                        break;
                }
        }
}
```

# LABEL

- Label can be given to coding blocks in Java

```
int[] xs = {1, 11, 22, 3, 4, 66, 77, 2, 100, 212};
int[] ys = {3, 30, 50, 23, 99};
loop1:
for(int x: xs){
        for(int y: ys){
                if(x + y > 40){
                        System.out.println("Invalid");
                        break loop1;
                }
        }
}
```

# LABEL

```
for(int i = 0; i < 3; i++){
        one:{
                two:{
                        System.out.println("i is " + i);
                        if(i == 1)break one;
                        if(i == 2)break two;
                        System.out.println("won't work");
                }
                System.out.println("After block two.");
        }
        System.out.println("After block one.");
}
```

# CONTINUE

- Continue: skip the current iteration of a loop.

```
int sum = 0;
int[] xs = {1, 5, 2, 8, 10, 11, 22};
for(int i = 0; i < xs.length; i++){
      if(xs[i] % 2 == 1) continue;
      sum += xs[i];
}
```

- What does this piece of code means?

# CONTINUE WITH LABEL

- What will happen here?

```
int[] xs = {1, 11, 22, 3, 4, 66, 77, 2, 100, 212};
int[] ys = {3, 30, 50, 23, 99};
loop1:
for(int x: xs){
        for(int y: ys){
                if(x + y > 40){
                        System.out.println("Invalid");
                        continue loop1;
                }
        }
}
```

# STRING

- One of the most important data structure in Java.
  - Strings are objects in Java, not primitive type.

- Construct a String:
- `String str = new String("Happy"); // like an object`
- `String str2 = new String(str); // from another string`

- Alternatively
- `String str = "Happy";`

# STRING METHODS

- Useful methods that operate on String:
- `boolean equals(Object str)//` return true, if they contains the same character sequence
- `int length() //` return the number of characters
- `char charAt(int index) //` return character at a specified index
- `int compareTo(String str) //` comparison based on Unicode of each character
- `int indexOf(char ch/ string str) //` return the index of the first occurrence of the given character or substring
- `int lastIndexOf(char ch/ string str) //` last index of..

- You can find more here
- https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

# IMMUTABLE STRING

- The contents of a String object are immutable.
- Example, replace returns a string resulting from replacing all occurences of oldChar in this string with newChar

```
String replace(char oldChar, char newChar)
String str1 = “Apple”;
str1.replace(’p’, ‘b’); // will it change the value of str1?
String str2 = str1.replace(’p’, ‘b’);
```

# IMMUTABLE STRING

- The contents of a String object are immutable.
- Example, replace returns a string resulting from replacing all occurences of oldChar in this string with newChar

```
String replace(char oldChar, char newChar)
String str1 = "Apple";
str1.replace('p', 'b');
String str2 = str1.replace('p', 'b');
```


replace


Once String "Apple" is created, it cannot be changed.

```
str1 = str1.replace('p', 'b');
// update the reference, "Apple" Garbage Collected
```

What if we need to manipulate a string in several steps? StringBuffer, StringBuilder

# STRING AND CHAR[]

- Both String and char[] represent a collection of characters, are they the same?

```
String str1 = "Happy";
for(char c : str1){
      ...
} // Will it work?

char[] cs1 = {'a', 'b', 'c'};
char[] cs2 = {'d', 'e', 'f'};
char[] cs3 = cs1 + cs2; // Will it work?
```

# STRING VS CHAR[]

- Other differences:

- Data type?
- Immutable?
- Build-in functions?
- Accessing each character?
- Conversions?

# STRING VS CHAR[]

- Data type: Single data type vs collections
- Immutable: Immutable vs mutable
- Build-in functions: String has a lot of build-in functions, char[] not.
- Accessing each character: charAt() vs var_name[index]
- Conversions:

```
String s = "Happy";
char[] cs = s.toCharArray();
cs = {'H', 'A', 'P', 'P', 'Y'};
s = new String(cs);
```

# TYPE CONVERSION

- Can we read integer and floating numbers from the command-line?

```
java Args 1 2
```

- We want 1 and 2 as integer number!

- Automatic type conversion:
  - Two types are compatible.
  - The destination type is larger than the source type.
  - E.g., byte to int, int to long, long to double, …
- Cast: an instruction to the compiler to convert one type into another

```
(target-type) expression
```

# TYPE CONVERSION

```
double x, y;
// …
int z = (int) (x / y);
```

- Narrowing conversion: information might be lost.
- E.g., information lost when we convert long to short

- Example:  CastDemo

- How to convert string into integer, double, …

# TYPE WRAPPER

- Type Wrapper: classes that encapsulate the primitive types.
- Primitive types: are not objects, e.g., cannot be passed by reference.
- Double, Integer, Float, …

- Numeric wrappers provide methods to convert a string into corresponding number.
- Double.parseDouble(String)
- Integer.parseInt(String)
- Short.parseShort(String)
- …
- Boolean values:
- Boolean.parseBoolean(String)