CM 10227: Lecture 8b

Dr Rachid Hourizi and Dr. Michael Wright

November 17, 2016

Java In/Out: In

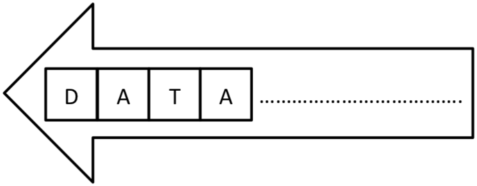
- To this point, we have looked at ways in which
  - Information can be passed to a program as part of an instruction to run (command line parameters) and
  - Objects can pass information to (and receive information from) other Objects
- We have not, however, looked at the ways in which the user can interact with our code whilst it is running
- The following slides will introduce the basics of passing Java input to a running program from a command line or console
- (Note that receiving user input via a graphical user interface is handled separately and will be covered in Semester 2)

- A good starting point for discussion of Input and Output is to consider the idea of a stream
- In contrast to the Arrays we saw earlier in the course, streams are (at least potentially) unlimited sequence of data
- As far as a running Java program knows, the data coming from a user's keyboard or a file
  - Is not of fixed length
  - And could (in theory at least) be infinite
- Similarly the amount of data that could (again in theory) be sent to a screen, printer or output file could also be infinite

- With that in mind, Java provides us with both input and output streams that we can use to
  - Read from a stream of input and
  - Write a stream of output to a designated destination (the screen, a file etc)
- Collectively, those input (I) and output (O) streams are described as IO streams
- This week, we will take a first look at input streams

- As usual with Java classes, we do not need to know how IO streams provide the functionality that they do
- We don't, for example, need to know how they
  - capture of data from a source (e.g. the keyboard)
  - transmit data to a final destination (e.g. the screen)
- We simply have to know and be able to use the interface that those classes provide in order to
  - Read data into a Java program and
  - Write data from it

- From http://tutorials.jenkov.com/java-io/streams.html:
  - Java IO streams are flows of data you can either read from, or write to.
  - IO Streams are typically connected to a data source, or data destination, like a file, network connection etc.
  - A stream has no concept of an index of the read or written data, like an array does.
  - Nor can you typically move forth and back in a stream, like you can in an array
  - A stream is just a continuous flow of data.


D
A
T
A

- We have already introduced three of the streams that are available to Java at runtime
  - Standard in
  - Standard out
  - Standard error
- Java calls these three streams
  - System.in
  - System.out
  - System.err

- The 3 streams System.in, System.out, and System.err are common sources or destinations of data
- These 3 streams are initialized by .. Java when (it) starts up, so you don't have to instantiate them yourself (although you can exchange them at runtime).
- Most commonly used is probably System.out for writing output to the console from console programs

- System.out
  - System.out is a PrintStream.
  - System.out normally outputs the data you write to it to the console.
  - This is often used from console-only programs like command line tools.
  - This is also often used to print debug statements of from a program (though it may arguably not be the best way to get debug info out of a program).

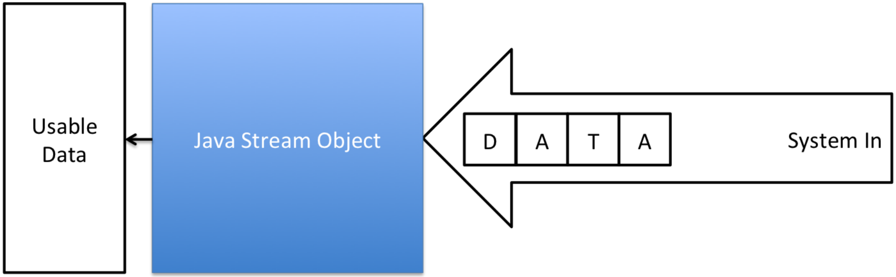
- System.err
    - System.err is also a PrintStream.
    - System.err works like System.out except it is normally only used to output error texts.
    - System.err often prints to the same location as System.out (the screen)
    - Some programs (like Eclipse) will show the output to System.err in red text, to make it more obvious that it is error text.

```
public class ExampleCode {
        public static void main(String[] args) {

        //printing to System.out
        System.out.println("Desired␣Output");

        //printing to System.err
        System.err.println("Informtion␣about␣an␣error");

    }
}
```

```
both outputs print to the screen:

rachidhourizi$ java Examplecode
Desired Output
Informtion about an error that has occurred
```

- System.in
    - System.in is an InputStream which is typically connected to keyboard input.
    - In applications with GUI the input to the application is given via the GUI. This is a separate input mechanism from Java IO.

- A group of Java classes take IO streams (e.g. Standard in, Standard our and Standard error) as input and return that data in more manageable chunks


Usable Data
Java Stream Object
D A T A
System In

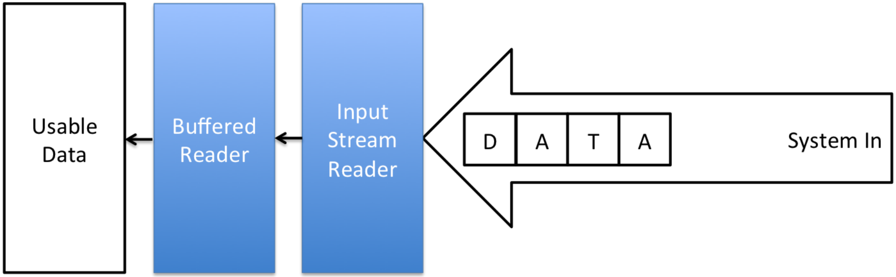
Different Classes return data in different forms:

- Some IO classes allow us to access data byte by byte (e.g. InputStream)
- Others give each line of data as a String (e.g. BufferedReader)
- And others help us to interpret ("parse") the incoming data (e.g. Scanner)

- We will look at the family of IO classes in more detail in the coming weeks
- At this point, however, we simply need to be able to
  - read data from an input stream
  - and pass that data to the rest of our code (e.g. SRPN)

- Passing data from System.in to a Bufferedreader requires us to take two steps:
  - First we have to create an instance of the InputStreamReader class (an InputStreamReader Object) and attach it to System.in
  - this can be done by calling the InputStreamReader constructor and passing it System.in as a parameter:
    - new InputStreamReader(System.in)
  - the InputStreamReader Object will take the stream of information coming from Standard.in and passes it to BufferedReader one char at a time

- Second, we have to call the BufferedReader constructor and pass it the InputStreamReader Object as a parameter
  - BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
- as a reult, we have a chain of Java IO objects:
- the InputStreamReader Object will take the stream of information coming from Standard.in and passes it to BufferedReader one char at a time
- BufferedReader will take that stream of chars and buffer them (store multiple chars ready for processing), presenting us with a String containing all the chars up to the point that the user presses enter on the keyboard


Usable Data
Buffered Reader
Input Stream Reader
D
A
T
A
System In

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class BufferedReaderX {
   public static void main(String[] args) throws Exception {

          //create String to hold output from BufferedReader
          String s;

      // create InputStreamReader Object
      // attach it to the stream of data coming from System.in
      // and use it to create a new BufferedReader Object
      BufferedReader br =
              new BufferedReader(new InputStreamReader(System.in));

      //ask user to make a first input
      System.out.println("Please enter something");

      //print out the user's last input and ask for further input
      while ((s=br.readLine()) != null) {
            System.out.println("You entered " + s
                 + " Please enter something else");
      }
   }
}
```

- At this point, you know how to read information in from System.in (i.e. from the user) one String at a time
- The example on the previous slide simply prints each String s to screen
- But for SRPN, you will need to look more closely at each incoming String
- Identify the different pieces of infomration within it
- i.e. operators and operands
- store those operators and operands somewhere
- ... and then perform the mathematical operations that they describe

- The following slides will give you some assistance as you think about how to write SRPN
- They will not, however, provide you with solutions that you can use “As-is” in your coursework
- If you cannot work out how to construct the SRPN using these building blocks, ask in the labs, post to programming1 etc.

- It is tempting to start by passing the information from System.in to a “Scanner”
- Scanner is an extremely helpful pre-written Class which allows you to extract the information in the String provided by System.in (or a BufferedReader) as one or more Integers, Floats etc.
- In order to use that class, you would set up a “chain” of Java IO Objects similar to the ones on the diagram above
- System.in – Scanner – Your Code
- or
- System.in – BufferedReader – Scanner – Your Code

```
import java.util.Scanner;
class ScannerTest{
 public static void main(String args[]){
   Scanner sc=new Scanner(System.in);
   System.out.println("Enter␣your␣name");
   String name=sc.next();
   System.out.println("Enter␣an␣Integer");
   int myNumber=sc.nextInt();
   System.out.println("Your␣name␣is:␣"+name +"Your␣number␣is:␣"+myNum
   sc.close();
 }
}
```

```
 Enter your name
 Rachid
 Enter an integer
 100
Your name is:Rachid Your number is:100
```

- Note the use of an import statement (somewhat similar to the include statements we saw in C)
- note also the use of the .next() method to pull out the next block of text from System.in
- note also the use of the .nextint() method to pull out the next integer from System.in

- You should note, however, that Scanner requires a recognisable separator ("delimiter") to exist between the chunks of information that it processes
- By default, Scanner expects white space (i.e. a space or a tab etc.) to be used to separate on piece of information from another
- though you can instruct a Scanner Object to look for other delimiters (e.g. commas)
- **This is a problem if the chunks of information ("tokens") being fed to your program are not separted by white space**
- e.g. Scanner would not interpret "4+3" (no spaces) as an int followed by an operator followed by another int
- it would see only a single String

- you can get some way into the SRPN exercise using a Scanner to interpret input
- but will need to adopt an alternative approach if you want to pass all of our automated tests

- You might want to do exactly that (Start with a Scanner and then change the way your code reads input) as part of an iterative programming approach

- An alternative approach is to read in each line of input as a String (using BufferedReader - see the example in a earlier slide)
- and then to analyse that String for yourself

- In order to do that, you may want to take advantage of methods defined by the String class e.g.:
  - charAt(int); //returns the character at the given index (position) in a String
  - getNumericValue(char) // returns the value of the given char interpreted as an int

```
String element = "el5";
int x = Character.getNumericValue(element.charAt(2));
System.out.println("x=" + x);
```

```
x=5
```

- For (large) numbers represented by mulitple characters, you may want to extract a part of the input string ("substring") before interpreting the value

```
import java.io.*;
public class Test {

  public static void main(String args[]) {
    String s1 = new String("HelloWorld");
    String s2 = new String(s1.substring(1,4));

    System.out.println(s2);
  }
}
```

```
ello
```

- One final step is to convert the String (or substring) provided by BufferedReader and convert it to an int (for use in the mathematical calculations performed by SRPN)
- this can be with the Integer.parseInt() method
  - int foo = Integer.parseInt("1234");
- This code snippet would leave us with variable foo holding the value 1234

- You may want to check that you have a String that can be converted to an Int before using ParseInt()
- this can be done with the Character.isDigit(char) method e.g.
    - boolean check = Character.isDigit(char);
- This method takes in a char
- ...returns true if that Char can be interpreted as an int
- ...and returns false if the input is not a char that can be interpreted as an int
- Note that you will have to check each char individually (using a loop and charAt()) to see whether the String that you want to check contains a multi-digit integer

- At this point, it is important to know to know that it is **not** possible to describe all elements in a Java program using Class and Object diagrams:
- In fact, Java defines two very different categories of data type:
- primitive types
  - int, char, boolean etc.
- object types
  - ArrayList, LinkedList, TicketMachine

- Object types originate from classes.
    - Some of which are predefined and
    - Some of which we can write ourselves
- The primitive types are non-object types.
    - Have no constructors, accessors, mutators
- All primitive types are predefined by Java.
- An aside: This means that Java is not a completely object oriented language

```
//asignement of a primitive to a variable
//without a call to a constructor
int a = 6;

//asignment of an Object to a variable
//using a constructor call
TicketMachine t = new TicketMachine(500);
```

- There are eight primitive data types supported by Java
  - byte (8-bit singed integer)
  - short (16-bit signed integer)
  - int (32-bit signed integer)
  - long (62-bit signed integer)
  - float (32-bit floating point number)
  - double (64-bit floating point number)
  - boolean (single bit true/false)
  - char (16-bit Unicode character)

- There are situations in which Java requires the use of an Object
- With this in mind, Java provides “wrapper classes”;
  - Classes which, when instantiated **do** provide objects
  - And hold the same values as the primitives used to create them

- Wrapper classes can be thought of as a way to make an Object from a primitive
- Wrapper classes mostly have the same name as the underlying primitive
- But the first letter is capitalized
  - The wrapper class for byte is therefore Byte
  - boolean is Boolean
  - and so on
- Two notable exceptions are
  - int: The wrapper class for int is Integer
  - char: The wrapper class for char is Character

Definition (Boxing)

Converting a primitive to a (wrapper) object
e.g. an int to Integer
is called boxing. Converting an Integer to int is called unboxing

Definition (Unboxing)

Converting a (wrapper) object to a primitive
e.g. an Integer to an int
is called boxing. Converting an Integer to int is called unboxing

- Wrapper classes have constructors, just as you would expect of any other class

```
//primitive
int a=7;

//constructing an Integer wrapper object using that primitive
Integer b = new Integer(a);
```

- Wrapper classes also provide accessors
- So using them as you would use primitives is poor practice
- But using them with an accessor method is acceptable

```
int a = 7;
Integer b = new Integer(5);

//intValue returns the value held
//in the Integer object as an int
a = a + b.intValue()
```

- Note: these accessors return primitives
  - e.g. in the example above, the returned value is an int (primitive)

- Wrapper classes are immutable, so they dont provide mutators
- Note however that Java does handle **some** boxing and unboxing (conversion to and from primitives) automatically so it will, in some circumstances look as a program is using the value contained within a wrapper object without an accessor

```
//example of automatic unboxing
Integer a=3;
Integer b=3;
a+=b;
System.out.println(a);
```

- Usefully, the wrapper classes also include accessor methods which return the value held as primitives of another type
- E.g. the Integer wrapper class provides the floatValue() method to return the underlying int as a float primitive:

```
float a = 7.0f;
Integer b = new Integer(5);

//floatValue() method returns the value held
//in the Integer object as a float
a = a + b.floatValue();

//print result
System.out.println(a);
```

```
Output is: 12.0
```

- The wrapper classes also include **static** methods which return the value represented by a String as a boxed object
- E.g. the Integer wrapper class provides a method to return the value represented by a String as an int primitive

```
String s ="10";
int a;

a= Integer.parseInt(s);
a=a+1;
System.out.println(""+a);
```

```
Output is: 11
```

- Static methods are provided by a class not by an object of that class
- This means that we dont need to create an instance of Integer (an Integer object) to use parseInt()
- We will return to the idea of static methods later in the course