F28DA11
Command Line Java Survival Guide

Phil Trinder and David H. Marwick

Java with BlueJ

- “BlueJ is a Java development environment explicitly for teaching introductory object-oriented programming”
  - Java has a steep learning curve
- BlueJ allows easy piecemeal testing of methods
  - very useful for development
- Running a Java program means allowing it to control what is happening

Java without BlueJ (App E)

- BlueJ hides functionality needed to run Java programs
  - thus, easing the learning process
- To progress, you must learn some of this hidden functionality
- You must write code
  - to create objects of your classes, and
  - to create appropriate interaction between these objects

Java Execution

- In BlueJ, to execute a program, typically you create an object of a control class. For example, in the address book programs (Ch 12)
  - an **`AddressBookDemo`** object is created, and
  - an appropriate method is invoked from that object
- This is eased by right-clicks within the BlueJ environment
- But what is actually happening?

Java Execution

- An object is created by invoking a constructor of the class. The hidden code is:

  `addressB1 = new AddressBookDemo();`

- Then we invoke the method **`showInterface()`** by right clicking on the object **`addressB1`**. The hidden code is

  `addressB1.showInterface();`

- How can we execute this code if BlueJ does not exist?
  - We have to tell the Java SDK where to start
  - As no object can exist before the program starts execution, we must point to a class method in the control class

Control Class

- A control class is used to start and control the operation of the program
- Typically, it is a small class containing class variables and methods
- It **must** include the method **`main`** with the signature:

```
public static void main(String[] args);
```

- There will be no created object of this class
  - What is a ***class method***? ***class variable***?

AddressBookDemo

- The control class for the address book projects is **AddressBookDemo**
- The contents of this class is now

```
public class AddressBookDemo{
  private static AddressBook book;
  private static AddressBookTextInterface
                                    interaction;
  public static void main(String[] args);
} //Note all variables and methods are static
```

- The program starts by executing **main**

main()

- **main** contains the code to start the program, including creating any objects necessary – for example:

```
public static void main(String[] args) {
  ContactDetails[] sampleDetails = {
    new ContactDetails("david", "08459 100000", "address 1"),
       ...
    new ContactDetails("ruth", "08459 800000", "address 8"),
  };
  book = new AddressBook();
  for(ContactDetails details : sampleDetails) {
        book.addDetails(details);
  }
  interaction = new AddressBookTextInterface(book);
  interaction.run();
}
```

- Compare this with the existing code of the class **AddressBookDemo** using the text-based interface

Executing using `main`

- This can all be achieved within BlueJ
- After compilation, execution is simply right clicking on **`AddressBookDemo`** and invoking the method **`main`**
  - **No object creation of `AddressBookDemo` is necessary**
    - Other objects are created within the program itself
  - No parameters are necessary
- However, Java programs must be able to run without using the BlueJ environment

Command line execution

- A (compiled) Java program can be run from a command line (in both Unix and Windows) by typing the command:

  `java AddressBookDemo`

  – This causes the Java virtual machine to start executing the method **`main`** in the control class **`AddressBookDemo`**

  – if **`main`** is not found, you get the error

`Exception in thread "main" java.lang.NoSuchMethodError: main`

  i.e. you can run only the control class

- The parameter of **`main`** enables data to be passed in the command (Command Line Arguments)

Compiling a Java program

- The Java compiler expects to find a text file of Java source code
- This can be created using any text editor
  - Notepad, Emacs, vi, etc (or even BlueJ)
  - Beware, do not use a word processor
- To compile using a command line:

```
javac AddressBookDemo.java
```

Java Files

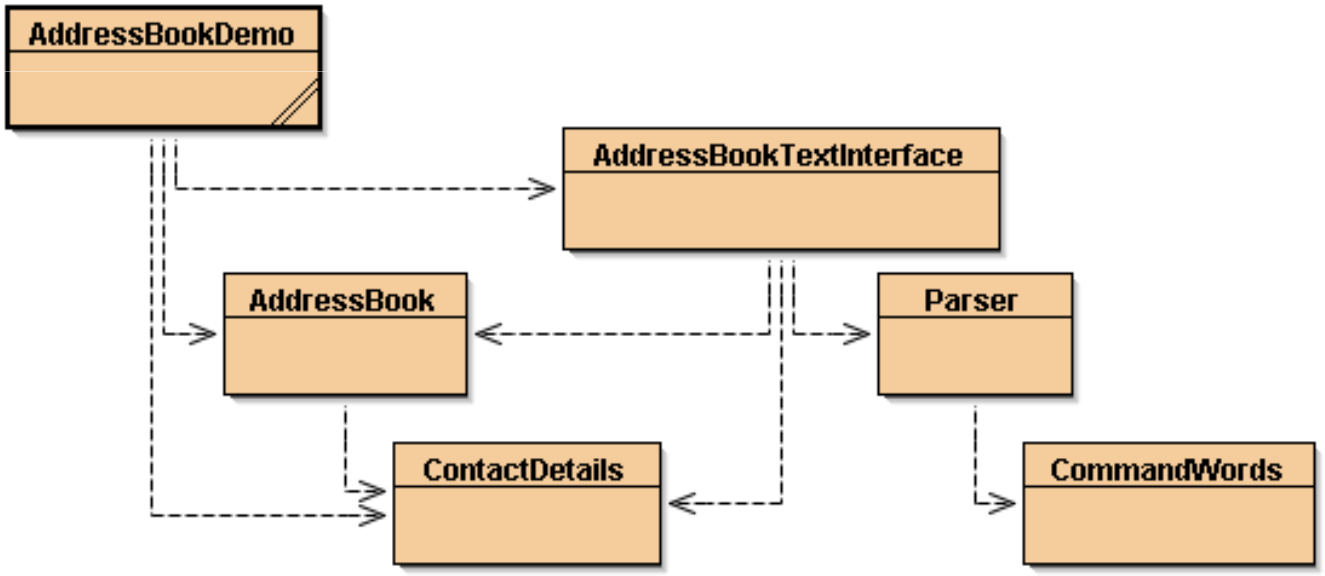
- The Java source code must be stored in a text file with the name of the class and the extension **`.java`**
- The Java compiler creates code files with the extension **`.class`** – one for each class
- Note that
  - to compile, the **`.java`** extension is used
  - to run, the **`.class`** extension is **not** used
- BlueJ has other files used only within BlueJ

Summary

- The control class is a *separate* class which **must** have a class method **`main`**
  - **`main`** must exist
  - **`main`** must be **`public`**
  - **`main`** must be **`static`** (class method)
  - **`main`** must have a String array parameter
  - To run, only **`main`** can be invoked

Class diagram for address-book

- Note that
  - the class diagram does not change
  - **`main`** creates objects and invokes the run method in the interface class



Simple I/O in Java

- In BlueJ, input and output are often done using the interface
- Stand alone Java programs must use the Java classes to achieve appropriate i/o
- We will now look at how to read data from the keyboard and display data on the screen

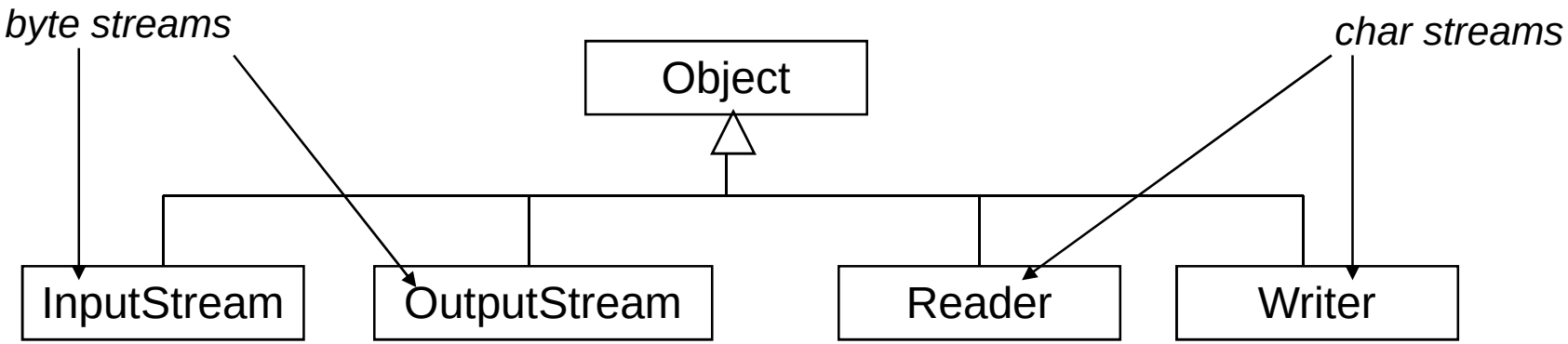
Java Input/Output

- In Java, data is read from & written to **streams**
- A stream is an abstraction that produces or consumes data
- To access real data, it is attached to a physical device
- It always behaves in the same manner regardless of the physical device used:
  - Keyboard
  - Disc file
  - Network socket
- Thus, stream classes (and methods) apply to all devices

Types of Stream

- There are two types of stream
- **Character streams** read and write Unicode characters
  - they are used for handling character data
- **Byte streams** read and write bytes (an 8-bit signed integer
  - they are used for handling binary data
- As all input is byte-oriented, the character stream classes convert bytes to characters automatically

Stream Hierarchy



- **`InputStream`** and **`OutputStream`** are the abstract superclasses for all byte I/O streams
- **`Reader`** and **`Writer`** are the abstract superclasses for all character I/O streams

Standard Streams

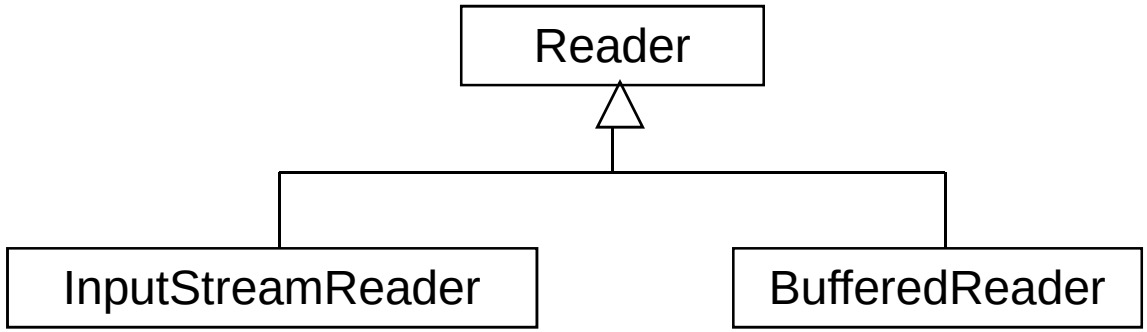
- The standard streams are:
  - the standard input stream, normally the keyboard
  - the standard output stream, normally the screen
  - the standard error stream, normally the screen
- The standard streams are based on the basic streams
  - **`System.in`** – an **`InputStream`** object
  - **`System.out`** } **`PrintStream`** objects, a
  - **`System.err`** } a subclass of **`OutputStream`**

Example: Sum 10 numbers

```
int sum, i, number;
sum = 0;
for (i = 0;  i<10; i=i+1){
   number =
      Integer.parseInt(System.in.readln());
   sum = sum + number;
}
System.out.println("Sum = "+sum);
```

- This code would not work because **`System.in`** is an **`InputStream`** object and does not recognise line terminators
  - It reads bytes only

Character Streams - `Reader`

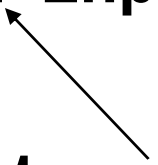


- **`InputStreamReader`** is the main class underlying the reading of character data
- **`BufferedReader`** is the class usually used for reading character data

Reading Characters

- The main constructors of these classes are:

```
public InputStreamReader(InputStream input);
public BufferedReader(Reader input);
```



- Note that **`System.in`** is an **`InputStream`** object – reading bytes
- **`InputStreamReader`** has methods which takes bytes and converts them to characters
- So a **`BufferedReader`** object normally builds on an **`InputStreamReader`** object

Reading from `System.in`

```
InputStreamReader keys =
    new InputStreamReader(System.in)
BufferedReader keyboard =
    new BufferedReader(keys)
```

OR

```
BufferedReader keyboard =
    new BufferedReader(new
        InputStreamReader(System.in))
```

`BufferedReader`

- The reason for using a **`BufferedReader`** object is to use the method

`public String readLine() throws IOException`

- This reads a line of characters from the input stream (excluding the line terminator)
- A **line** is a structure within a character stream

Example: Sum 10 numbers

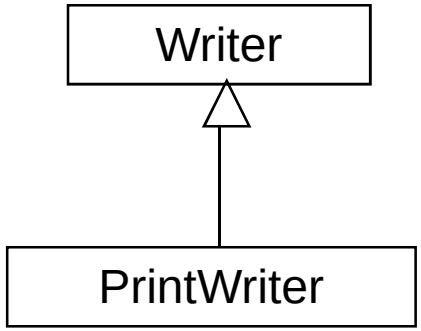
```
int sum, i, number;
BufferedReader keyboard =
    new BufferedReader(new
        InputStreamReader(System.in));
sum = 0;
for (i = 0;  i<10; i=i+1){
   number = Integer.parseInt
         (keyboard.readLine().trim());
   sum = sum + number;
}
System.out.println("Sum = "+sum);
```

`System.out`

- **`System.out`** is a **`PrintStream`** object
- **`PrintStream`** is a subclass of **`OutputStream`**, a byte stream
  - **`PrintStream`** permits output of characters
  - The statement

    `System.out.println("Sum = "+sum);`
  - causes **`sum`** to be converted to a **`String`** and the whole parameter is output to the screen on one line

Writer



- The main constructors are:

```
public PrintWriter(OutputStream output,
                   boolean flush);
public PrintWriter(Writer output,
                   boolean flush);
```

`PrintWriter` Methods

- **`println()`**buffers and displays the parameter followed by a new line
- **`print()`** buffers without a new line (and so, does not display)
- **`flush()`** causes the buffer to be flushed (and so, displayed)
  - the second parameter in the constructor specifies if the stream is flushed after **`println`**

Output

- Note that **`System.out`** is an **`OutputStream`** object – writing bytes
- So a **`PrintWriter`** object can be connected to **`System.out`** (using the first constructor on slide 32)

```
PrintWriter screen =
      new PrintWriter(System.out, true);
```

- Note that the **`PrintWriter`** methods can also be used with **`System.out`**

Output

- We can invoke the methods of this stream to display our output - for example

```
screen.print("Enter an integer: ");
screen.flush();
```

- Output values are buffered prior to output
- In order to display the data, the buffer must be **flushed**
  - a new line also flushes the buffer

Example with keyboard & screen

```
public static void main(String[] args) throws IOException{
   int sum, i, number;
   BufferedReader keyboard =
     new BufferedReader(new InputStreamReader(System.in));
   PrintWriter screen =  new PrintWriter(System.out, true);
   sum = 0;
   for (i = 0;  i<10; i=i+1){
     screen.print("Enter an integer: ");
     screen.flush();
     number = Integer.parseInt(keyboard.readLine().trim());
     sum = sum + number;
   }
   screen.println("Sum = "+sum);
}
```

Wrapper Classes

- Wrapper classes provide an object wrapper around primitive data types
- They are used to provide methods to convert from a **`String`** to a primitive type
- **`String`**s that do not represent a value of a primitive type throw an exception

Wrapper Classes (cont.)

- two **`Integer`** methods widely used are:

```
public static int parseInt(String s) throws
NumberFormatException;

public int intValue();
```

- **`parseInt`** is a method which does **not** operate on an **`Integer`** object – a class method

```
n = Integer.parseInt(keyboard.readLine().trim());
```

- **`intValue` does** operate on an **`Integer`** object – an instance method

```
n = new Integer(keyboard.readLine().trim()).intValue();
```

Wrapper Classes (cont.)

- **`Integer`** also has two public class constants **`MAX_VALUE`** and **`MIN_VALUE`,** which give the range of **`ints`**
- there are similar wrapper classes for **`Double`, `Long`, `Float`, `Short`,** and **`Byte`,** e.g.

```
public static double parseDouble(String s)
throws NumberFormatException;

public double doubleValue();
```

Useful `String` methods

- **`String`** methods can be used to manipulate the input string
- e.g. **`trim()`**removes all leading and trailing white space in a **`String`** object. Thus,

```
keyboard.readLine().trim()
```

returns a trimmed **`String`** object

- To access a single **`char`** value:

```
char ch = keyboard.readLine(). charAt(0);
```

which returns the character at position 0 (left-most) of the trimmed **`String`** object**`trim().`**

Summary

- main method: **`public static void main(String[] args);`**
- Compile: **`javac myprog.java`**
- Run: **`java myprog`**
- Java I/O:
  - Streams, standard streams like **`System.in`**
  - Reading from streams, e.g. keyboard
  - Writing to streams, e.g. screen