Unit 8:

Exception Handling and Basic GUI

Object-Oriented Programming (OOP)
CCIT 4023, 2025-2026

U8: Exception Handling and Basic GUI

- Exception Handling
  - **`try-catch`** Control Flow
  - Propagating Exceptions
  - Examples of Exception Handling
- Basic GUI, with A Sample **`Swing`** Example
  - **`JFrame`**, **`JPanel`**
  - **`JLabel`**, **`JTextField`** and **`JButton`**
- Package, JAR, and Javadoc comments

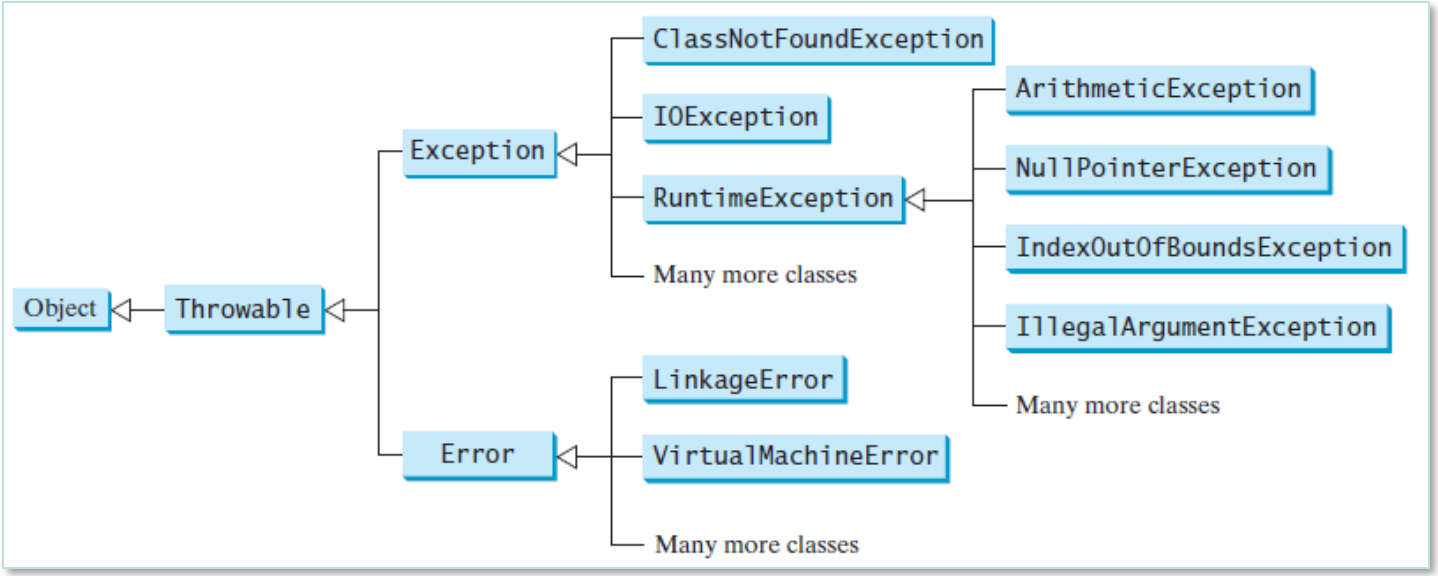
Exception Handling

- Program correctness guarantees correct results for *valid inputs*
- What happens when the input is invalid? E.g.
  - Wrong type of argument is passed to a method
  - Invalid input value is entered
- A mechanism called **exception handling** can be used to improve the program's robustness
- An ***exception*** represents an error condition that may occur during the normal course of program execution
- When an exception occurs, we say an exception is ***thrown***, and the normal sequence of flow is terminated, then the ***exception-handling routine*** is executed
- When the matching exception-handling code is executed, we say the thrown exception is ***caught***

What is Exception Handling ?

- Java exceptions are objects of classes under the `Throwable` class inheritance hierarchy. Two subclasses of `Throwable` class:
  - `Error` represents serious problems not expecting to be handled / caught.
  - `Exception` represents conditions expecting to be handled / caught.

- There are two basic types of exceptions:
  - Unchecked (or Runtime) exceptions, under classes `RuntimeException` and `Error`, and their subclasses
  - Checked exceptions, the rest other than unchecked exceptions



Types of Exceptions

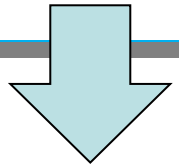
- **Checked** exceptions are ***checked at compilation time***.
  - As checked exceptions are checked and validated at the compilation time, this type of exceptions must be handled (with `try-catch` or propagation).
  - Compilation error occurs if this is not handled.
  - E.g. `IOException, SQLException, ClassNotFoundException`, etc.
- **Unchecked** exceptions (also called runtime exception) represent defects in the program (bugs), and they are ***unchecked at compile time*** and are detected only at runtime.
  - As unchecked exceptions are not checked at the compile time, no compilation error occurs if this exception type is not handled.
  - However, an exception may still occur during program execution.
  - `NullPointerException, ArrayIndexOutOfBound, ArithmeticException, NumberFormatException,` etc.

Not Catching an Unchecked Exception

- Below shows an example case that an exception leads to an error message of invalid input (e.g. User input "ten", blank) during runtime

```
String inputStr;
int    age;

inputStr = JOptionPane.showInputDialog(null, "Age:");
age = Integer.parseInt(inputStr); // may cause exception
```



```
Exception in thread "main" java.lang.NumberFormatException:
For input string "ten"
        at java.lang.Integer.parseInt(Integer.java:449)
        at java.lang.Integer.parseInt(Integer.java:499)
        at Unit4Sample1.main(Unit4Sample1.java:27)
```

Catching an Exception

- Exception is handled by a set of **`try-catch`**:
  - A `try` block, with one (or more) `catch` block directly after the `try` block.
  - There could be more than one statement in each block.
  - Exception (if any) will end `try` block, then execute the related `catch` block

try-block

catch-block

```
// ... LAST section
try {
    String inputStr;
    int    age;
    inputStr = JOptionPane.showInputDialog(null, "Age:");
    age = Integer.parseInt(inputStr); // may cause exception
    JOptionPane.showMessageDialog(null, "Your age: "+age);
} catch (NumberFormatException e){
    JOptionPane.showMessageDialog(null, "Invalid!\n"
        +  "Please enter digits only!");
}
// ... NEXT section
```

General Control Flow of `try-catch`

Case 1: NO EXCEPTION

Complete the `try`-block, skip `catch`-block

```
// ... LAST section
try {
   String inputStr;
   int    age;
   inputStr = JOptionPane.showInputDialog(null, "Age:");
   age = Integer.parseInt(inputStr); // may cause exception
   JOptionPane.showMessageDialog(null, "Your age: "+age);
} catch (NumberFormatException e){
    JOptionPane.showMessageDialog(null, "Invalid!\n"
        +  "Please enter digits only!");
}
// ... NEXT section
```

Case 2: With EXCEPTION

Skip the rest of `try`-block from where exception comes, and execute the `catch`-block

```
// ... LAST section
try {
   String inputStr;
   int    age;
   inputStr = JOptionPane.showInputDialog(null, "Age:");
   age = Integer.parseInt(inputStr); // may cause exception
   JOptionPane.showMessageDialog(null, "Your age: "+age);
} catch (NumberFormatException e){
    JOptionPane.showMessageDialog(null, "Invalid!\n"
        +  "Please enter digits only!");
}
// ... NEXT section
```

Getting Information from Exception

- There are two methods we can call to get or display information about the thrown exception:
  - method `getMessage()` : returns the detail message string of this exception
  - method `printStackTrace()` : displays this exception and its backtrace

```
try {
    //...
} catch (NumberFormatException e){
    //...
    System.out.println(e.getMessage());
    e.printStackTrace();
}
```

Multiple `catch` Blocks

- A single `try-catch` statement can include multiple `catch` blocks, one for each type of exception
  - Exception (if any) will end `try` block, and then execute *only* the related `catch` block

```
try {
    //...
    age = Integer.parseInt(inputStr); //MAY cause NumberFormatException
    //...
    val = abc / xyz; // MAY cause ArithmeticException
    //...
} catch (NumberFormatException e){// type of NumberFormatException
    //...
} catch (ArithmeticException e ){// type of ArithmeticException
    //...
}
```

General Control Flow with Multiple `catch` Blocks

Case 1: NO EXCEPTION

Complete the `try`-block, skip all `catch`-blocks

```
// ... LAST section
try {
     //...
     age = Integer.parseInt(inputStr);
     //...
     val = abc / xyz;
     //...
} catch (NumberFormatException e){
     //...
} catch (ArithmeticException e ){
     //...
}
// ... NEXT section
```

Case 2: With EXCEPTION

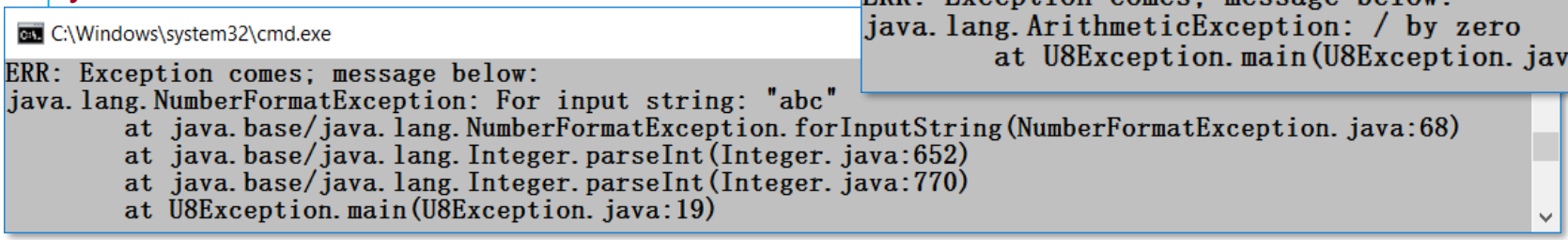
Skip the rest of `try`-block from where exception comes, and execute only the related `catch`-block (skip all other `catch`-blocks)

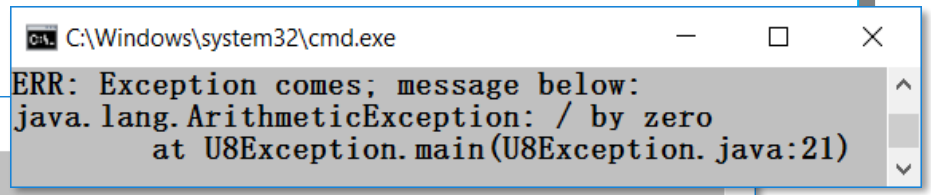
```
// ... LAST section
try {
     //...
     age = Integer.parseInt(inputStr);
     //...
     val = abc / xyz;
     //...
} catch (NumberFormatException e){
     //...
} catch (ArithmeticException e ){
     //...
}
// ... NEXT section
```

A Lazy Way to Handle Multiple Exceptions (with One `catch` Block)

- Instead of using multiple `catch` blocks handle different types of exceptions, we may use one superclass exception to catch all of its subclass exceptions in one `catch` block.
  - However, this only provides one handling action (to all related exceptions)

```
try {
  //...
  age = Integer.parseInt(inputStr);// cause NumberFormatException
  //...
  val = abc / xyz; // cause ArithmeticException
  //... // may have more other Exceptions
} catch (Exception e){//Exception, superclass of other subclasses
  System.out.println(" ERR: Exception comes; message below:");
  e.printStackTrace();
}
```





The `finally` Block

- There are situations where we need to take certain actions regardless of whether an exception is thrown or not
  - We place statements that must be executed regardless of exceptions in the `finally` block
- Generally, a `try` statement with a `finally` block is executed by first executing the `try` block, then:
  - If execution of the `try` block completes normally, the `finally` block is executed (`try-finally`)
  - If execution of the `try` block does not complete, the corresponding `catch` block is executed if any, then the `finally` block (`try-catch-finally`)

ALL Cases:

`finally`-block must be executed, in both cases of with or without exception.

```
// ... LAST section
try {
     //...
     age = Integer.parseInt(inputStr);
     //...
} catch (NumberFormatException e){
     //...
} finally {
     // MUST BE DONE...
}
// ... NEXT section
```

Throwing Exceptions

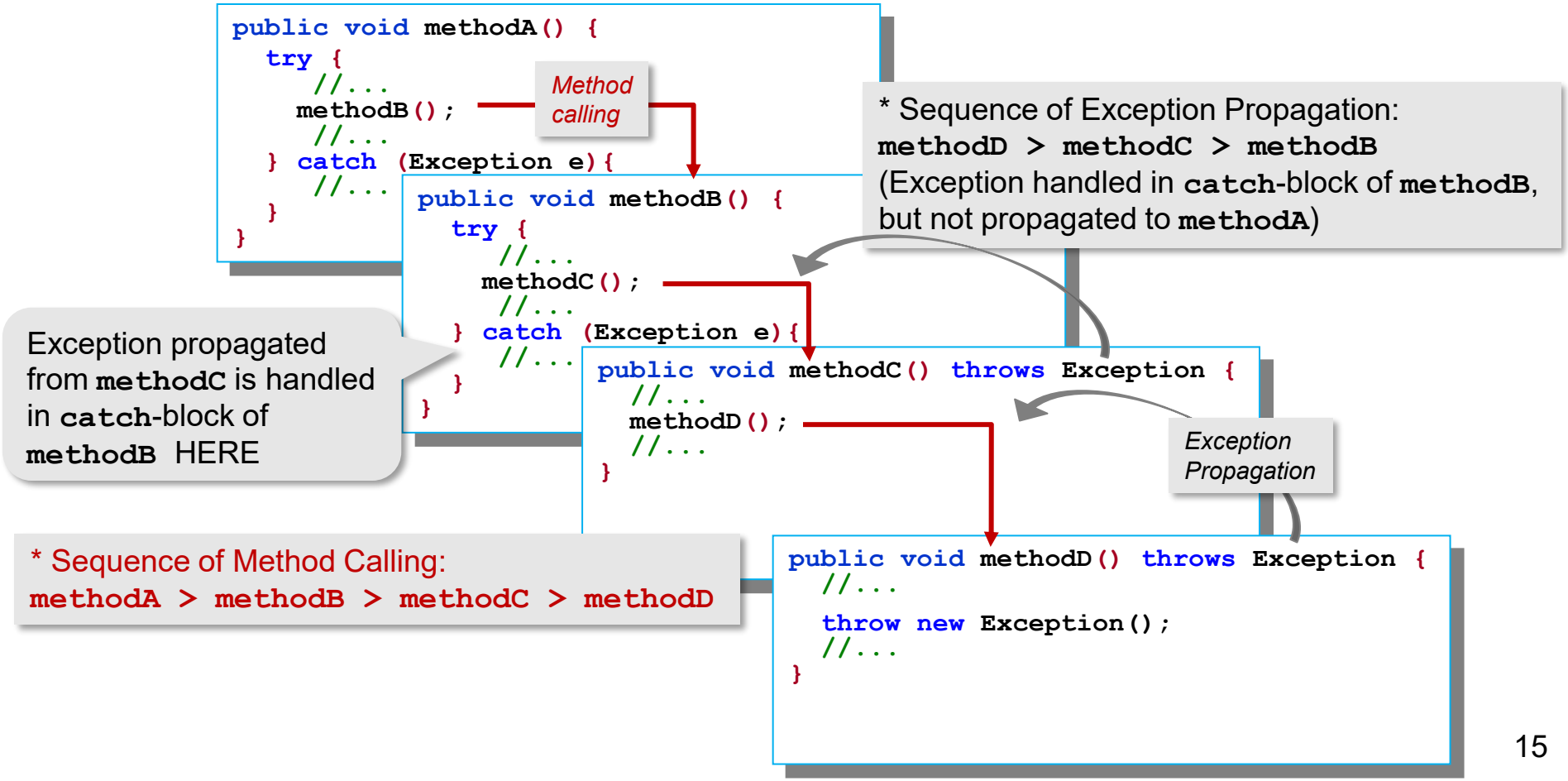
- The exception can also be thrown *explicitly* with our own codes, by using the keyword **`throw`**, e.g.

```
public int getAge( ) throws NumberFormatException,
                                    ArithmeticException {
   // may cause NumberFormatException below
   int age = Integer.parseInt(inputStr);
   // ...
   // throw an exception below, explicitly
   throw new ArithmeticException();
   return age;
}
```

Throw Exception Explicitly

Propagating Exceptions

- Instead of catching a thrown exception by using `try-catch`, we may propagate the thrown exception back to the caller of method
  - The method header includes the keyword **`throws`**, e.g.



Customized Exception

- Although less common, we can define/customize our own exception class.
- We may develop our own exception class (e.g. `CustomException` below) which inherits class `Exception` as its subclass.

```
public class CustomException extends Exception {
        //...
}
```

Reference: https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/Exception.html

An Example: Handling File I/O Exception

Reference Only

- It is basically identical to read from and write to text file, as to console.
  - Below shows simple example codes, including exception handlings:

```
// SimpleRWTxtFile.java: Read/Write with text file ////////////////
import java.util.Scanner;
import java.io.PrintStream;
import java.io.File;
public class SimpleRWTxtFile{  // File "in.txt", with 2-lines string expected
    public static void main(String[] args) {
        String inF = "in.txt"; // input File.  REQUIRED!!
        String outF = "out.txt"; // output File
        Scanner scanner=null;
        System.out.println("Read 2 string lines from file: "+inF);
        try {
            scanner = new Scanner(new File(inF)); // Scanner, from file
        } catch (Exception e) {}  // ONLY for demo, do nothing here!!
        String aStrLine1 = scanner.nextLine();  // scan a line as a string
        String aStrLine2 = scanner.nextLine();  // scan a line as a string

        System.out.println("String lines to be written, to file: "+outF);
        try {
            System.setOut(new PrintStream(outF)); // set out to a file
        } catch (Exception e) {}  // ONLY for demo, do nothing here!!
        System.out.println(aStrLine1);
        System.out.println(aStrLine2);
    }
}
```

Set scanner with a file, not default console/User

Set System out with a file, not default console

Exception

Reference Only

- The `parseInt()` method of the Class `Integer`

  ** Reference to the API document for details*

parseInt

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

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

`s` - a String containing the int representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

`NumberFormatException` - if the string does not contain a parsable integer.

Sample of Propagating Exceptions

Reference Only

parseInt

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

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

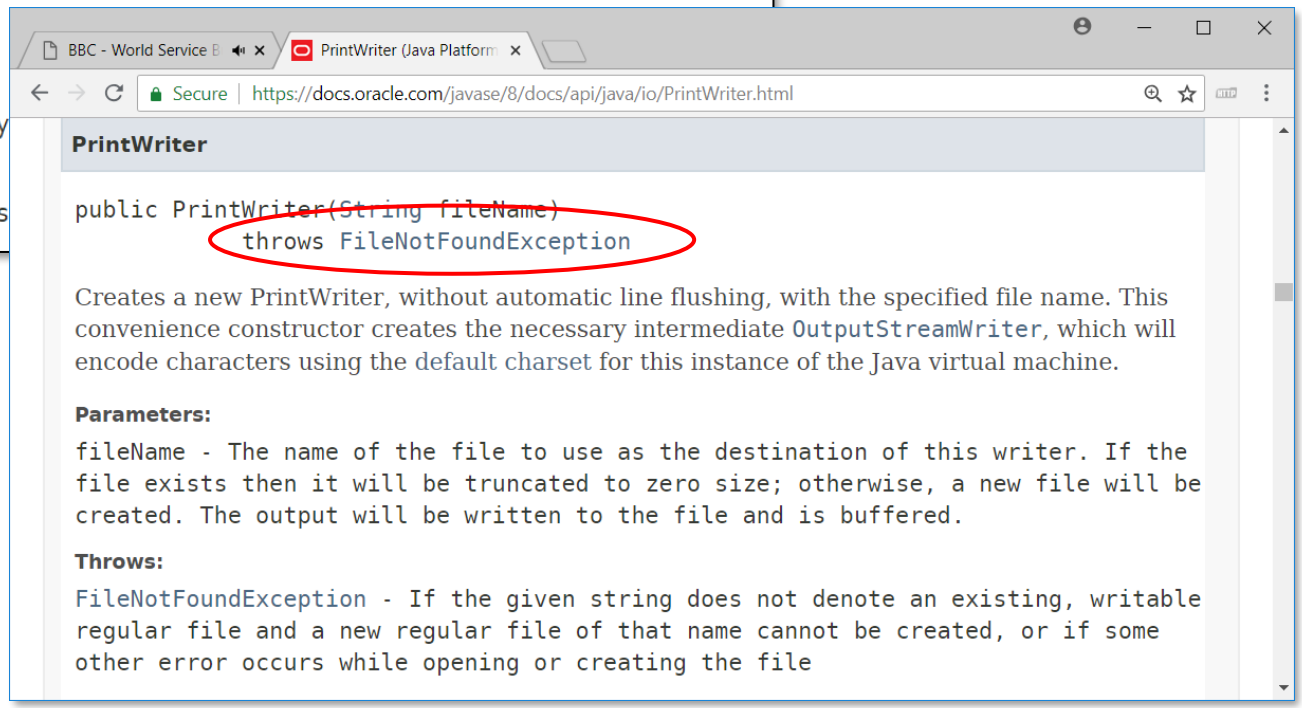
`s - a String containing the int`

Returns:

`the integer value represented by`

Throws:

`NumberFormatException - if the s`



PrintWriter

```
public PrintWriter(String fileName)
            throws FileNotFoundException
```

Creates a new PrintWriter, without automatic line flushing, with the specified file name. This convenience constructor creates the necessary intermediate `OutputStreamWriter`, which will encode characters using the default charset for this instance of the Java virtual machine.

Parameters:

`fileName - The name of the file to use as the destination of this writer. If the file exists then it will be truncated to zero size; otherwise, a new file will be created. The output will be written to the file and is buffered.`

Throws:

`FileNotFoundException - If the given string does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file`

Start GUI with A Sample `Swing` Example

- Use `JOptionPane` may only build some very simple Graphical User Interaction (GUI) applications
- For a more complicated application, we may define the layout and graphical user interface components with more features and techniques supported by Java, including
  - GUI Composition and Layout
  - Event Handling
  - Using Common GUI Components
- This section provides a sample example of Swing GUI-based application, using classes in the Swing package `javax.swing`
  - There are other packages often needed for developing GUI-based application, e.g. packages `java.awt` and `java.awt.event`

* *Remark:* More details in Java GUI development will be further discussed in later units.

Start GUI with A Sample **Swing** Example
(JFrame, JPanel, JButton, JTextField and JLabel)

Some classes in the Swing package to produce more complex GUI, e.g.:

1. Create a JFrame, for a main window

```
JFrame aF = new JFrame("Window Title");
```

2. Create a JPanel, for a container of other components

```
JPanel aP = new JPanel();
```

3. Create a JButton, for simple user input with button click

```
JButton aB = new JButton("Button Text");
```

4. Create a JTextField, for inputting text information

```
JTextField aTF = new JTextField (20);
```

5. Create a JLabel, for showing information (text or image)

```
JLabel aL = new JLabel("A Line of Message");
JLabel imgL = new JLabel((new ImageIcon("aPhoto.jpg"));
```

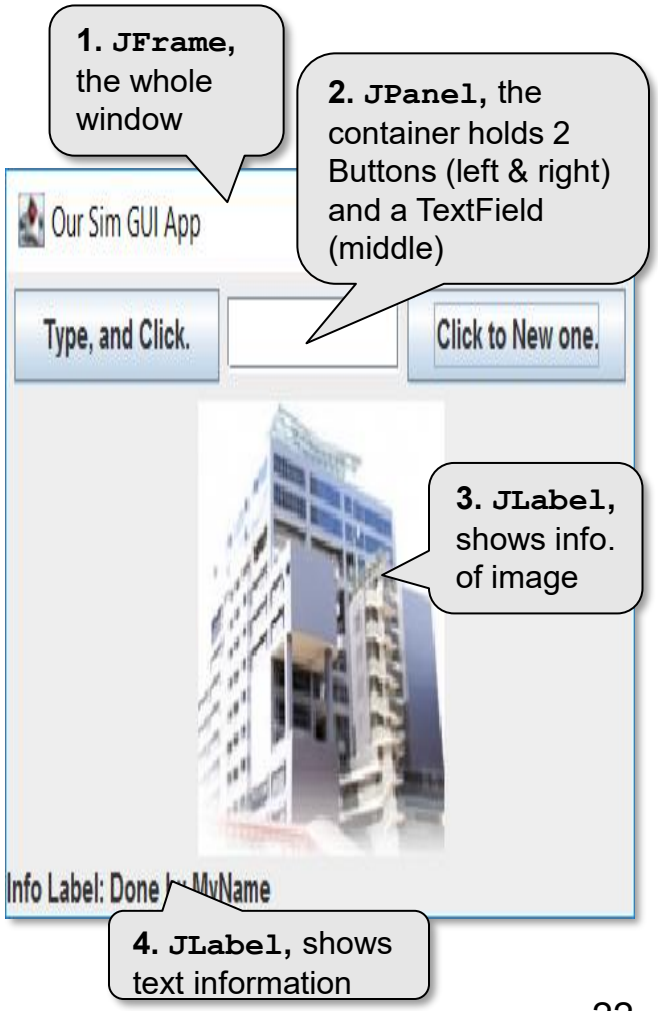
Start GUI with A Sample **`Swing`** Example

(`JFrame`, `JPanel`, `JButton`, `JTextField` and `JLabel`)

This section gives am example to illustrate how to make a simple GUI program

- *Details of GUI will be given in later units.*

1. The whole Window (with `JFrame`)
2. Top (NORTH) Menu Panel (`JPanel`), contains 2 buttons (`JButton`) and Text Field (`JTextField`)
   - Left-Button-Click leads to a dialog pops up showing what user has typed in text field
   - Right-Button-Click leads to a new SAME window pops up
3. Middle image, contained in a `JLabel`
4. Bottom (SOUTH) Panel
   - Shows text information in a `JLabel`



Start GUI with A Sample **Swing** Example
(JFrame, JPanel, JButton, JTextField and JLabel)

Our GUI program consists of:

1. A JFrame, as the main window, in a form of:

```
JFrame aGUIWin = new JFrame("Our Sim GUI App");
```

2. A JPanel, as the top (NORTH) panel of window, for containing of 2 buttons and a text field

```
JPanel northP = new JPanel();
```

- Create a JButton, for user button-click

```
JButton northRB = new JButton("Simply Click");
```

- Create a JTextField, for inputting text

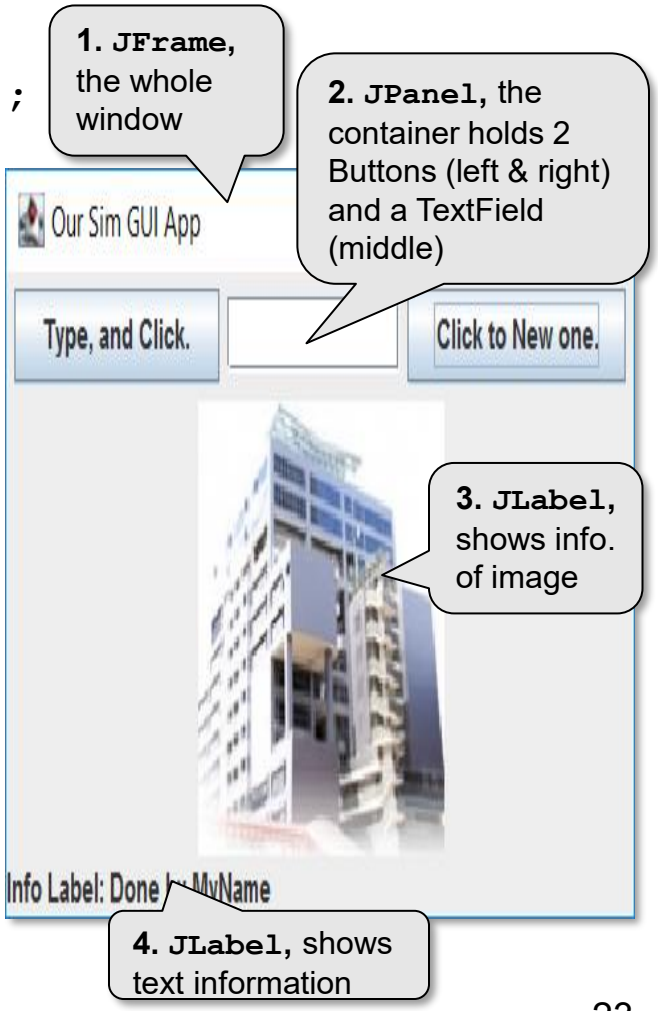
```
JTextField northTF = new JTextField(10);
```

3. Create a JLabel, for showing image

```
JLabel centerImgL = new JLabel(
                    new ImageIcon("cc.jpg"));
```

4. Create a JLabel, for showing text

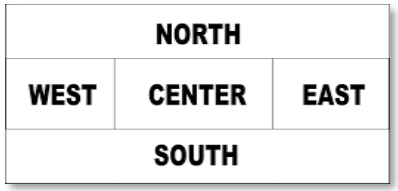
```
JLabel souchInfoL = new JLabel();
```



Start GUI with A Sample **`Swing`** Example
(`JFrame`, `JPanel`, `JButton`, `JTextField` and `JLabel`)

- Add an element to a `JFrame`, e.g.

  `aGUIWin.add(northP, BorderLayout.NORTH);`
  - By default, `JFrame` has a BorderLayout



- Similary, add an element to a `JPanel`, e.g. `northP.add(northLB);`
  - By default, `JPanel` has a `FlowLayout`, from left to right, top to bottom.

- For GUI components with user input, like button, codes for responding action must be handled (with event handling and interface), e.g.

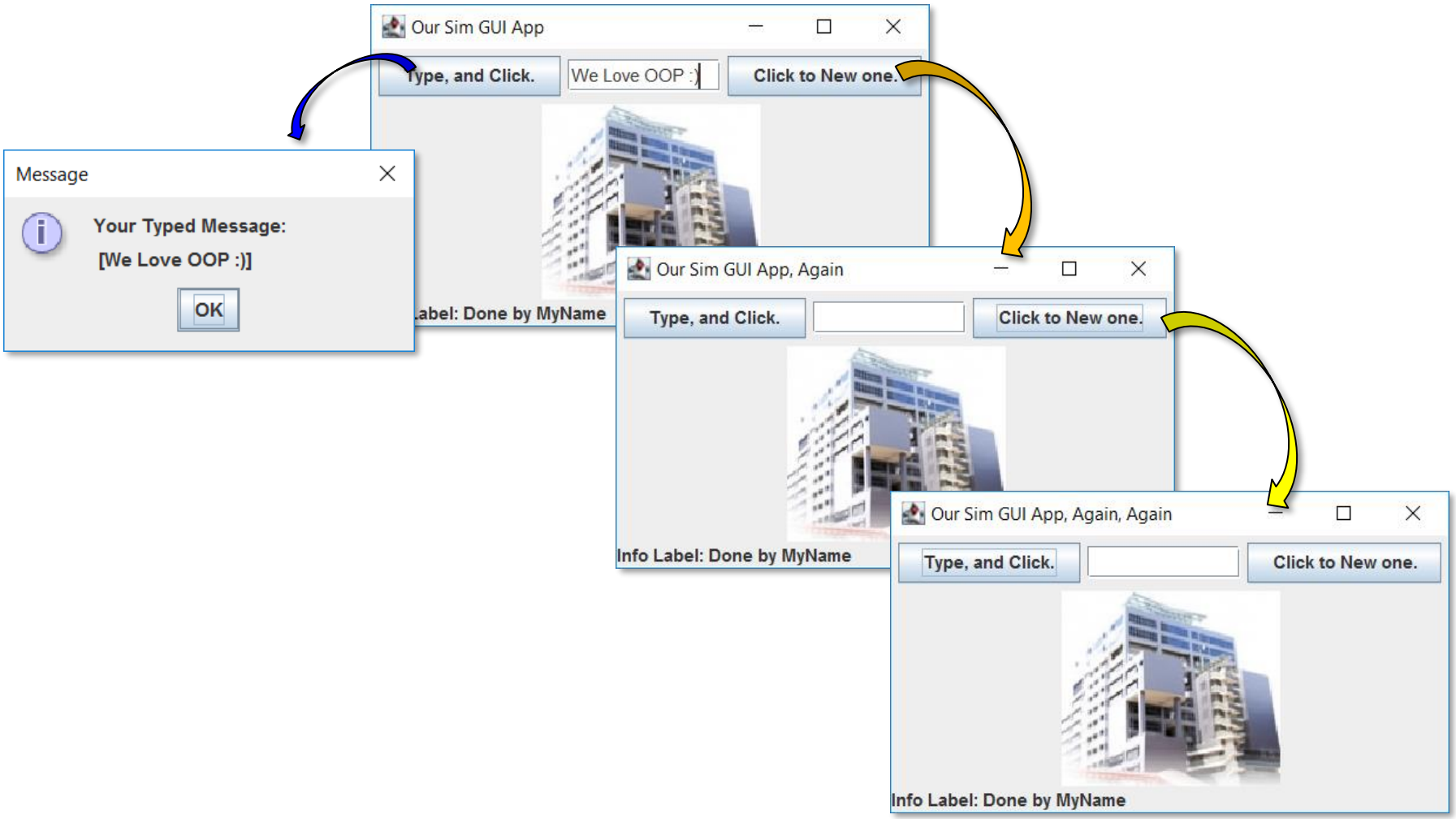
```
northLB.addActionListener(cEvent -> { //button click jumps here
        ////// code to let button-click respond HERE  //////
});
```

- Often the `JFrame` will be set with close window response, say exit the program, pack and set it visible. E.g.

```
// set JFrame close, to exit program
aGUIWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// resize to fit layouts of subcomponents; show window (JFrame)
aGUIWin.pack(); aGUIWin.setVisible(true);
```

Start GUI with A Sample **`Swing`** Example

(`JFrame`, `JPanel`, `JButton`, `JTextField` and `JLabel`)



Start GUI with A Sample **Swing** Example

(JFrame, JPanel, JButton, JTextField and JLabel)

```
/* SimGUIApp.java: A simple GUI-based application for demo, for U7 */
import javax.swing.*;               // import for all classes in specific package
import java.awt.*;
import java.awt.event.*;
public class SimGUIApp{  // class declaration, NOT using Anonymous class --> NO output U7GUI$1.class, etc.
    public static void genGUI(String fTitle){ // a method to generate GUI window
        JFrame aGUIWin = new JFrame(fTitle);                    // 1. main window JFrame
        JPanel northP = new JPanel();                            // 2. top panel
        JButton northLB = new JButton("Type, and Click.");       // 2.1 left button, at top
        JButton northRB = new JButton("Click to New one.");           // 2.2 right button, at top
        JTextField northTF = new JTextField(10);          // 2.3, textfield for typing; width size = 20
        JLabel centerImgL = new JLabel(new ImageIcon("cc.jpg")); // 3. a label, for showing image
        JLabel souchInfoL = new JLabel("Info Label: Done by MyName");  // 4. a label, at bottom for text info.
// constructing top panel, containing two components responding to button and textfield
        northLB.addActionListener(cEvent -> { // button-click response: show simple dialog
            JOptionPane.showMessageDialog(null,    // pop a dialog window
                    "Your Typed Message:\n ["+northTF.getText()+"]\n");
        });
        northRB.addActionListener(cEvent -> { // button-click response: show simple dialog
            genGUI(fTitle + ", Again"); // create and pop another window
        });
        northP.add(northLB);        northP.add(northTF); northP.add(northRB); // add elts to top panel
        aGUIWin.add(northP, BorderLayout.NORTH); // add panel to window (JFrame), to top NORTH
// constructing middle image label, and add it to window
        aGUIWin.add(centerImgL, BorderLayout.CENTER); // add image label to CENTER
// constructing bottom information label, and add it to window
        aGUIWin.add(souchInfoL, BorderLayout.SOUTH);  // add text info to SOUTH
// finalizing and setting the window JFrame, visible, etc.
        aGUIWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // close JFrame to exit program
        aGUIWin.pack();  aGUIWin.setVisible(true); // show the window (JFrame)
    }

    public static void main(String[ ] args) {
        genGUI("Our Sim GUI App");
        genGUI("Our Sim GUI App - 2");
    }
}
```

Package, JAR, and Javadoc comments
(Organizing Classes into a Java Package)

- Without proper organizing, for class A to use class B, their bytecodes must be located in same folder
  - This is not practical if we want to reuse programmer-defined classes in many different programs
  - To make classes and types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related classes and types into packages
- A ***package*** is a Java class library
  - A *package* is a grouping of related classes and types providing access protection and name space management
  - Without a package statement (esp. for small or temporary developments), an unnamed (default) package is attached

Creating and Using a Java Package

- **Naming a Package**
  - Package names should be selected with all lower case to avoid conflict with the names of classes
  - Commonly companies use their reversed Internet domain name to begin their package names, e.g.:

    `com.example.mypackage` for a package named `mypackage` created by a programmer at `example.com`

- **Creating a Package**
  - Put a package statement with the package name at the top of every source file that contains the types that we want to include in the package, e.g.

```
package com.example.mypackage; // package statement
import javax.swing.JOptionPane; // import statement
class HiWorld {  // class (named HiWorld) declaration
//..
```

Create, View and Compile with a JAR file

- The **Java™ Archive (JAR)** file enables you to bundle multiple files into a single archive file
- This JAR file often includes, associated classes of the package, resources (e.g. images), etc.
- To perform basic tasks with JAR files, we use the Java Archive Tool (the `jar` command) provided as part of the Java Development Kit (JDK)

Using javadoc Comments

- Many programmer-defined classes we designed are intended to be used by other programmers
  - It is, therefore, very important to provide meaningful documentation to the client programmers so they can understand how to use our classes correctly
- By adding **`javadoc` comments** to the classes we design, we can provide a consistent style of documenting the classes (the Official *Java Platform API Specification documentation*)
- Once the `javadoc` comments are added to a class, we can generate HTML files for documentation by using the `javadoc` command

Creating and Using a Java Package

- Most IDE for developing Java program may manage and associate a package automatically to the Java project (such as in NetBeans IDE)
- Managing Package Files with JDK commands
  - Compile: we may use `-d` option in `javac`, to compile with creating package folders and placing class files into it, e.g.

    ```
    javac -d .\ HiWorld.java
    ```

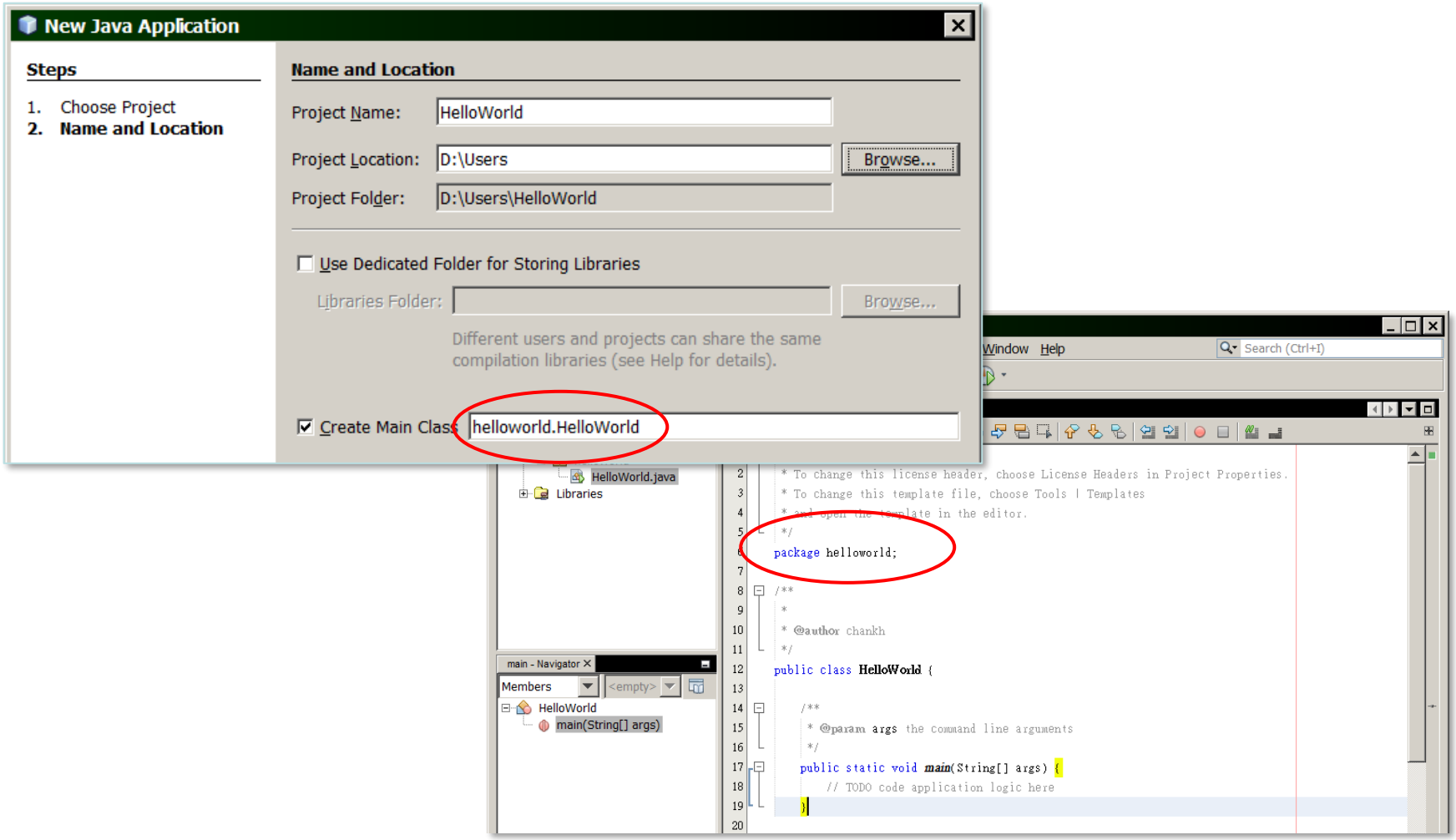
    - The command above will create relevant folders `.\com\example\mypackage` in the current directory, and the generated `HiWorld.class` will be placed there
  - Run: to run the `HiWorld` class from the current directory, we have to indicate the full package as:

    ```
    java com.example.mypackage.HiWorld
    ```

Create Java Package in IDE (NetBeans)

Reference Only

- IDE tools often create Java project with package, e.g. in NetBeans



Creating and Using a Java Package

- Using a public package member (class) from outside its own package, there are 3 ways.
- Below shows how to use the class `HiWorld`, outside its package `com.example.mypackage`
  - Referring to a member by Its full qualified name (without import statement), e.g.

    ```
    com.example.mypackage.HiWorld myHi;
    ```

  - Import a specific member

    ```
    import com.example.mypackage.HiWorld;
    ```

  - Import the entire package

    ```
    import com.example.mypackage.*;
    ```

Create, View and Compile with a JAR file

- Common JAR file operations:

Operation	Command
To create a JAR file	`jar cf jar-file input-file(s)`
To view the contents of a JAR file	`jar tf jar-file`
To extract the contents of a JAR file	`jar xf jar-file`
To extract specific files from a JAR file	`jar xf jar-file archived-file(s)`
To run an application packaged as a JAR file	`java -jar app.jar`

Run with a JAR file

- ***Create*** a JAR file (with Command Prompt):

```
jar cf <jar-file> input-file(s)
```

- ***View*** contents of a JAR file:

```
jar tf <jar-file>
```

- ***Compile*** Java file with JAR:

```
javac -cp <jar-file> MyClass.java
```

- ***Run*** JAR-packaged application "directly" (associated with a main class):

```
java -jar <jar-file>
```

- ***Run*** a specified class (containing the entry-point `main()` method) in a JAR (may or may not be the main class):

```
java -cp <jar-file> MyPackage.MyClass
```

JAR file in NetBeans IDE

- In standard project, NetBeans IDE builds a JAR file from project sources every time we run the **`Build`** command or the **`Clean and Build`** command
  - The JAR file is generated to the **`dist`** directory of your project folder
- On Windows platform, it is common to run executable **`jar`** on *double click*
  - This will basically run the main class with **`javaw`** command
- **`javaw`** command is identical to **`java`**, except that with **`javaw`** there is no associated console window.
  - Use **`javaw`** when you do not want a command prompt window to appear, e.g. run a pure GUI program.

javadoc for the sample class Fraction

- This is a portion of the HTML documentation for the Fraction class shown in a browser
- This HTML file is produced by processing the javadoc comments in the source file of the Fraction class

All Classes
Fraction

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Fraction

java.lang.Object
└ Fraction

public class Fraction
extends java.lang.Object

Constructor Summary

Fraction(int num, int denom)	

Method Summary

Fraction	add(Fraction frac)
static int	gcd(int num, int denom) Returns the greatest common divisor of two positive integers It is the largest integer that divides both without remainder.
int	getDenominator()
int	getNumerator()
void	setDenominator(int denominator)
void	setNumerator(int numerator)
Fraction	simplify()

javadoc Tags

- The `javadoc` comments begins with `/**` and ends with `*/`
- Special information such as the authors, parameters, return values, and others are indicated by the `@` marker

```
@author (classes and interfaces only)
@version (classes and interfaces only)
@param (methods and constructors only)
@return (methods only)
@exception
@see
@since
//...etc
```

Generate `javadoc`

Reference Only

1. Open the command prompt
2. Change directory to the folder that contains the java files
3. Type: `javadoc -d .\javadoc\ Fraction.java`

   (Requirement: The path of java is set)

- Detailed reference on how to use `javadoc` on Windows is located at: http://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/index.html

Example: javadoc Source

Reference Only

```
//...
/**
* Returns the greatest common divisor of two positive integers
* It is the largest integer that divides both without
remainder.
* @param num Numerator
* @param denom Denominator
*
* @return the greatest common divisor
*/
public static int gcd (int num, int denom) {
        if (denom == 0)
                return num;
        return gcd(denom, num%denom);
}
//...
```

this javadoc source will produce …

Example: javadoc Source


```
Loading source file Fraction.java...
Constructing Javadoc information...
Standard Doclet version 1.6.0_21
Building tree for all the packages and classes...
Generating .\javadoc\Fraction.html...
Generating .\javadoc\package-frame.html...
Generating .\javadoc\package-summary.html...
Generating .\javadoc\package-tree.html...
Generating .\javadoc\constant-values.html...
Building index for all the packages and classes...
Generating .\javadoc\overview-tree.html...
Generating .\javadoc\index-all.html...
Generating .\javadoc\deprecated-list.html...
Building index for all classes...
Generating .\javadoc\allclasses-frame.html...
Generating .\javadoc\allclasses-noframe.html...
Generating .\javadoc\index.html...
Generating .\javadoc\help-doc.html...
Generating .\javadoc\stylesheet.css...
```

Example: javadoc Output

Reference Only

gcd

```
public static int gcd(int num,
                      int denom)
```

Returns the greatest common divisor of two positive integers It is the largest integer that divides both without remainder.

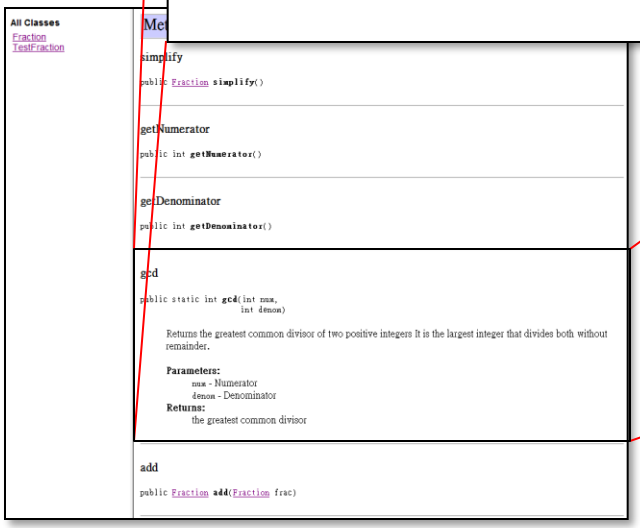
Parameters:

`num` - Numerator

`denom` - Denominator

Returns:

the greatest common divisor



More on javadoc Comments

Reference Only

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
}
```

References:
http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
http://www.oracle.com/technetwork/articles/java/index-137868.html

More on javadoc Comments

Reference Only

getImage

public Image getImage(URL url, String name)

Returns an Image object that can then be painted on the screen.
The url argument must specify an absolute URL. The name argument is a specifier that is relative to the url argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

url - an absolute URL giving the base location of the image.

name - the location of the image, relative to the url argument.

Returns:

the image at the specified URL.

See Also:

Image

Generate Javadoc in NetBeans

- To generate Javadoc documentation for a NetBeans project:
  - Select the project in the Projects window
  - Choose **`Run > Generate Javadoc (for Project)`**
- The IDE generates the Javadoc to the **`dist/javadoc`** folder in the project directory, and
  - then opens the index.html page in the IDE's designated web browser

References

- This set of slides is only for educational purpose.

- Part of this slide set is referenced, extracted, and/or modified from the followings:
  - Deitel, P. and Deitel H. (2017) “Java How To Program, Early Objects”, 11ed, Pearson.
  - Liang, Y.D. (2017) “Introduction to Java Programming and Data Structures”, Comprehensive Version, 11ed, Prentice Hall.
  - Wu, C.T. (2010) “An Introduction to Object-Oriented Programming with Java”, 5ed, McGraw Hill.
  - Oracle Corporation, “Java Language and Virtual Machine Specifications” https://docs.oracle.com/javase/specs/
  - Oracle Corporation, “The Java Tutorials” https://docs.oracle.com/javase/tutorial/
  - Wikipedia, Website: https://en.wikipedia.org/