UNIT – III

Exception handling and Multithreading-- Concepts of exception handling, benefits of exception handling, termination or resumptive models, exception hierarchy, usage of try, catch, throw, throws and finally, built in exceptions, creating own exception subclasses. String handling, Exploring java.util. Differences between multithreading and multitasking, thread life cycle, creating threads, thread priorities, synchronizing threads, inter thread communication, thread groups, daemon threads. Enumerations, autoboxing, annotations, generics.

**Exception Handling**

An exception is an abnormal condition that arises in a code sequence at run time. In other words run-time error.A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the java run-time system, or they can be manually generated by your code.

**Advantage of Exception Handling**

1. It allows you to fix the error.
2. It prevents the program from automatically terminating.

**Exception Types**

All the exception types are subclasses of the built-in class Throwable. Thus, Throwable is the top of the exception class hierarchy. This is as shown in the figure.

java.lang

Throwable

Error Exception

Runtime Exception Other Exception

Sub Classes of Runtime Exception sub classes

**General form of Exception Handling block**

try

{

// block of code to monitor for errors

}

catch (ExceptionType1 exOb)

{

// exception handler for ExceptionType1

}

catch (ExceptionType2 exOb)

{

// exception handler for ExceptionType2

}

…..

Finally

{

// block of code to be executed after try block ends

}

When there is an exception, the user data may be corrupted. This should be tackled by the programmer by carefully designed the program. For this, we should perform the following 3 steps

**Step 1**: The programmer should observe the statements in his program where there may be possibility of exception. Such statements should be written inside a try block. A try block looks like as follows

**try**

**{**

**Statement**

**}**

The greatness of try block is that even if some exceptions arises inside it, the program will not be terminated when JVM understands that there is an exception. It stores the exception details in an exception stack and then jumps into a catch bock. **In simple words to guard against and handle a run-time error. Simple enclose the code that you want to monitor inside the try block**

**Step2:**

The programmer should write the catch block where he should display the exception details to the user. This helps the user to understood that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this error. Catch block look like as follows

**catch (Exceptionclass ref)**

**{**

**Statements**

**}**

The **reference ref above** is automatically adjusted to refer to the exception stack where the details of the exception are available. So we can display the exception details using any one of the following

1. using print() or println() methods such as System.out.println(ref);
2. using printStackTrace() method of Throwable class, which fetches exception details from the exception stack and display them

The goal of most well constructed clause should be to resolve the exceptional condition and then continue on as if the error had never happened.

**Step :3**

Lastly the programmer should perform clean up operations like closing the files and terminating the threads. The programmer should write this code in the finally block.

**finally**

**{**

**Statements**

**}**

The specialty of finally block is that the statements inside the finally block are executed irrespective of whether there is an exception or not. This ensures that all the opened files are properly closed and all the running threads are properly terminated. So the data in the files will not be corrupted and user is at the safe side.

Performing the above three tasks is called ‘exception Handling remember in exception handling the programmer is not preventing the exception, as in many cases it is not possible. But the programmer is avoiding any damage that may happen to user data.

**throw clause**

using throw it is possible for our program to throw an exception explicitly.

The general form of throw is shown below

**throw ThrowableInstance;**

Here, **ThrowableInstance** must be an object of type Throwable or a subclass of Throwable.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace

**throws clause**

even if the programmer is not handling runtime exceptions, the java compiler will not give any error related to runtime exception. But the rule is that the programmer should handle checked exceptions. In case the programmer does not want to handle the checked exception he should throw them out using throws clause. Otherwise there will be an error flagged by the java compiler.

General form of a method declaration that includes a throws clause is:

***type method-name(parameter-list) throws exception-list***

***{***

***// body of method***

***}***

**Java’s Built-in Exceptions**

**Checked Exception:-**The Exception which are checked by the compiler for smooth execution of the program at runtime are called checked Exception.

**Unchecked Exception:-** The Exception which are checked at runtime by JVM is called Unchecked Exception

**Java’s Unchecked RuntimeException Subclasses Defined in java.lang**

1. ArithmeticException 🡪 Arithmetic error, such as divide-by-zero.
2. ArrayIndexOutOfBoundsException 🡪 Array index is out-of-bounds.
3. ArrayStoreException 🡪Assignment to an array element of an incompatible type.
4. ClassCastException 🡪Invalid cast.
5. EnumConstantNotPresentException🡪 An attempt is made to use an undefined enumeration value.
6. IllegalArgumentException 🡪Illegal argument used to invoke a method.
7. IllegalMonitorStateException 🡪Illegal monitor operation, such as waiting on an unlocked thread.
8. IllegalStateException 🡪Environment or application is in incorrect state.
9. NegativeArraySizeException🡪 Array created with a negative size.
10. NullPointerException 🡪Invalid use of a null reference.
11. NumberFormatException 🡪Invalid conversion of a string to a numeric format.

**Java’s Checked Exceptions Defined in java.lang**

1. ClassNotFoundException Class not found.
2. CloneNotSupportedException Attempt to clone an object that does not implement the Cloneable interface.
3. IllegalAccessException Access to a class is denied.
4. IllegalArgumentException Illegal argument used to invoke a method.
5. InstantiationException Attempt to create an object of an abstract class or interface.
6. InterruptedException One thread has been interrupted by another thread.
7. NoSuchFieldException A requested field does not exist.
8. NoSuchMethodException A requested method does not exist.

**Programs on Exception Handling**

**/\* demo program which causes abnormal termination of program\*/**

class Exam1

{

public static void main(String args[])

{

int d=0;

int a=42/d;

}

}

**Output**

C:\exception>javac Exam1.java

C:\exception>java Exam1

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Exam1.main(Exam1.java:6)

**/\* Alternative way of above program which uses method \*/**

class Exam2

{

static void subroutine()

{

int d=0;

int a=10/d;

}

public static void main(String args[])

{

Exam2.subroutine();

}

}

**Output**

C:\exception>javac Exam2.java

C:\exception>java Exam2

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Exam2.subroutine(Exam2.java:6)

at Exam2.main(Exam2.java:10)

**/\* program which uses try catch to avoid abnormal termination\*/**

class Exam3

{

public static void main(String args[])

{

int a,d;

try

{

d=0;

a=42/d;

System.out.println("This will not be printed");

}catch(ArithmeticException e)

{

System.out.println("Division by Zero");

}

System.out.println("After catch Statement");

}

}

**Output**

C:\exception>javac Exam3.java

C:\exception>java Exam3

Division by Zero

After catch Statement

**/\* demo program which uses multiple ctach statement which handle multiple exceptions\*/**

class MultiCatch

{ public static void main(String args[])

{

try

{

int a = args.length;

System.out.println("a = " + a);

int b = 42 / a;

int c[] = { 1 };

c[42] = 99;

} catch(ArithmeticException e)

{ System.out.println("Divide by 0: " + e); }

catch(ArrayIndexOutOfBoundsException e)

{ System.out.println("Array index oob: " + e); }

System.out.println("After try/catch blocks.");

}

}

**Output**

C:\exception>javac MultiCatch.java

C:\exception>java MultiCatch

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks.

**/\* program which uses nested try statements \*/**

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

{ try

{ int a = args.length;

int b = 42 / a;

System.out.println("a = " + a);

try {

if(a==1)

a = a/(a-a);

if(a==2) { int c[] = { 1 };

c[42] = 99;

}

} catch(ArrayIndexOutOfBoundsException e) { System.out.println("Array index out-of-bounds: " + e); }

} catch(ArithmeticException e) { System.out.println("Divide by 0: " + e); }

}

}

**Output**

C:\exception>java NestTry

Divide by 0: java.lang.ArithmeticException: / by zero

C:\exception>

**/\* demo program which uses throw keyword \*/**

class ThrowsDemo {

static void throwOne() {

System.out.println("Inside throwOne.");

throw new IllegalAccessException("demo");

}

public static void main(String args[]) {

throwOne();

}

}

// This is now correct.

class ThrowsDemo {

static void throwOne() throws IllegalAccessException {

System.out.println("Inside throwOne.");

throw new IllegalAccessException("demo");

}

public static void main(String args[]) {

try {

throwOne();

} catch (IllegalAccessException e) {

System.out.println("Caught " + e);

}

}

}

**/\* demo program on finally \*/**

// Demonstrate finally.

class FinallyDemo {

// Through an exception out of the method.

static void procA() {

try {

System.out.println("inside procA");

throw new RuntimeException("demo");

} finally {

System.out.println("procA's finally");

}

}

// Return from within a try block.

static void procB() {

try {

System.out.println("inside procB");

return;

} finally {

System.out.println("procB's finally");

}

}

// Execute a try block normally.

static void procC() {

try {

System.out.println("inside procC");

} finally {

System.out.println("procC's finally");

}

}

public static void main(String args[]) {

try {

procA();

} catch (Exception e) {

System.out.println("Exception caught");

}

procB();

procC();

}

}

**/\* demo program on throws keyword \*/**

// This program contains an error and will not compile.

class ThrowsDemo {

static void throwOne() {

System.out.println("Inside throwOne.");

throw new IllegalAccessException("demo");

}

public static void main(String args[]) {

throwOne();

}

}

// This is now correct.

class ThrowsDemo {

static void throwOne() throws IllegalAccessException {

System.out.println("Inside throwOne.");

throw new IllegalAccessException("demo");

}

public static void main(String args[]) {

try {

throwOne();

} catch (IllegalAccessException e) {

System.out.println("Caught " + e);

}

}

}

**Creating User-defined Exception**

Sometimes the built-in exception in java are not able to describe a certain situation. In such cases, like the built-in exception, the user can also create his own exception which are called ‘user-defined exceptions.. the following steps are followed in creation of user-defined exception.

1. The user should create an exception class as a sub class to Exception class. Since all exceptions are subclasses of exception class, the user should also make his class a subclass to it. This done as

**Class MyException extends Exception**

1. The user can write a detail constructor in his own exception class. He can use it. In case he does not want to store any exception details. If the user does not want to create an empty object to his exception class, he can eliminate the default constructor.

**MyException(){}**

1. The user can create an parameterized constructor with string as a parameter. He can use this to store exception details. He can call super class constructor from this and send the string.

**MyException(String str)**

**{**

**super(str);**

**}**

1. When the user wants to raise his own exception. He should create an object to his exception class and throw it using throw class

**MyException me=new MyException(“Exception details”);**

**Throw me;**

**Example program**

To understand how to create user-defined exception. Let us write a program in which we are creating our own exception class MyException. In this program we are taking the details of account numbers, customer names and balance amounts in the form of three arrays. Then in main() method, we display these details using a for loop. At this time, we check if in any account the balance amount less than the minimum balance to be kept in the account. If so, then MyException is raised and a message is displayed “Balance amount is less”

class MyException extends Exception

{

private static int accno[]={1001,1002,1003,1004,1005};

private static String name[]={"Raja Rao","Rama rao","Subba rao","appa rao","Laxmi Devi"};

private static double bal[]={10000,12000,5500,99.00,55};

MyException()

{

}

MyException(String str)

{

super(str);}

public static void main(String args[])

{

try

{

System.out.println("AccNo"+"\t"+"Customer"+"\t"+" Balance");

for(int i=0;i<5;i++)

{

System.out.println(accno[i]+"\t"+name[i]+"\t"+bal[i]);

if(bal[i]<1000)

{

MyException me=new MyException("balance amount is less");

throw me;

}

}

}catch(MyException me)

{

me.printStackTrace();

}

}

}

**C:\Users\svit\Desktop\java Notes>javac MyException.java**

**C:\Users\svit\Desktop\java Notes>java MyException**

**AccNo Customer Balance**

**1001 Raja Rao 10000.0**

**1002 Rama rao 12000.0**

**1003 Subba rao 5500.0**

**1004 appa rao 99.0**

**MyException: balance amount is less**

**at MyException.main(MyException.java:26)**

**Chained Exceptions**

Chained *Exception* helps to identify a situation in which one exception causes another *Exception* in an application.**For instance, consider a method which throws an *ArithmeticException*** because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero.The method will throw the *ArithmeticException* to the caller. The caller would not know about the actual cause of an *Exception*. Chained *Exception* is used in such situations.

To allow chained exceptions, two constructors and two methods were added to Throwable. The constructors are shown here:

**Throwable(Throwable causeExc)**

**Throwable(String msg, Throwable causeExc)**

In the first form, causeExc is the exception that causes the current exception. That is, causeExc is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception.

The chained exception methods added to Throwable are getCause( ) and initCause( ).

**Throwable getCause( )**

**Throwable initCause(Throwable causeExc)**

The getCause( ) method returns the exception that underlies the current exception. If there is no underlying exception, null is returned. The initCause( ) method associates causeExc with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. Thus, you can call initCause( ) only once for each exception object.

**// Demonstrate exception chaining.**

class ChainExcDemo {

static void demoproc() {

// create an exception

NullPointerException e =

new NullPointerException("top layer");

// add a cause

e.initCause(new ArithmeticException("cause"));

throw e;

}

public static void main(String args[]) {

try {

demoproc();

} catch(NullPointerException e) {

// display top level exception

System.out.println("Caught: " + e);

// display cause exception

System.out.println("Original cause: " +

e.getCause());

}

}}

**Multitasking**

Executing several tasks simultaneously is the concept of multitasking

There are 2 types of multitasking

1. Process based multitasking
2. Thread based multitasking

**Process based multitasking**

Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called process based multitasking.

Example while typing a java program in the editor we can able to listen mp3 audio songs simultaneously at the same time we can download a file from the internet. All these are executing simultaneously and independent of each other. It is process based multitasking. This type multitasking best suitable for OS level.

**Thread based multitasking**

Executing several tasks simultaneously where each task is a separate independent part of the same program. Such type of multitasking is called thread based multitasking.This type of multitasking is best suitable for programmatic level and each independent part is called thread.

Java provides inbuilt support for multithreading by introducing a rich library (Thread, Runnable, ThreadGroup etc) ***Whether it is process based or thread based the main objective of multitasking is to improve performance, by reducing response time.***

The main application areas of multithreading are developing video games and multimedia graphics etc.

**Difference b/w process based and thread based multitasking.**

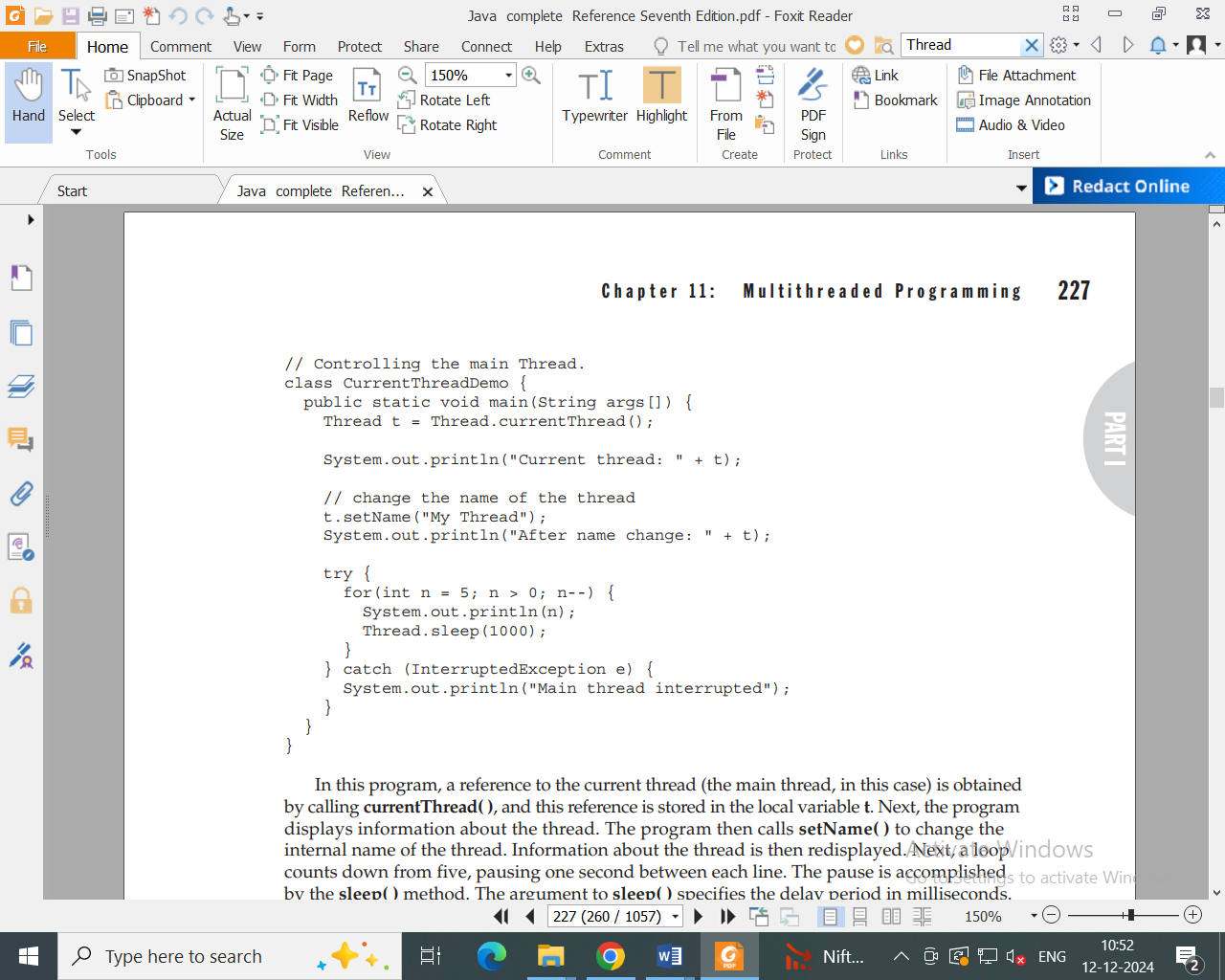
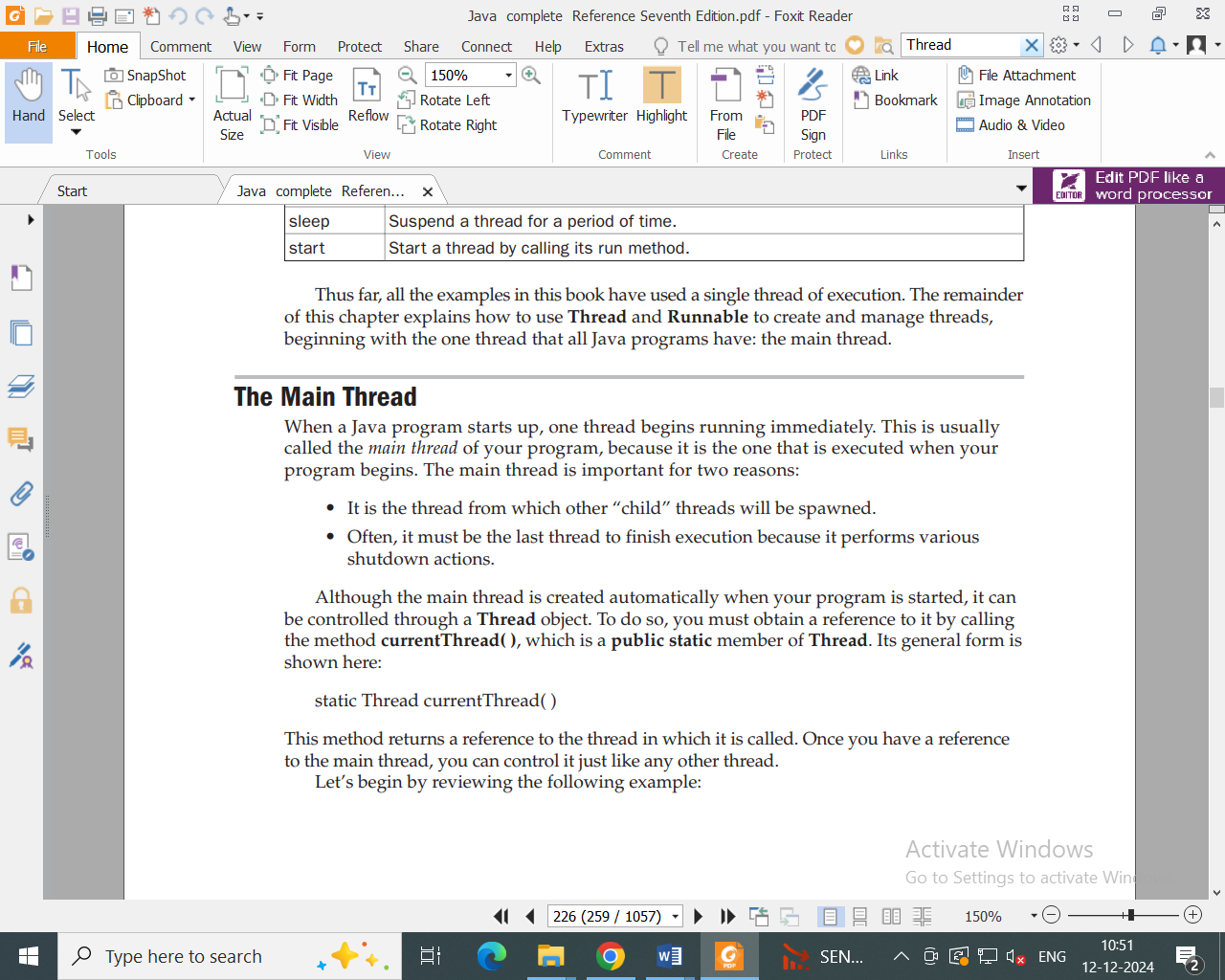
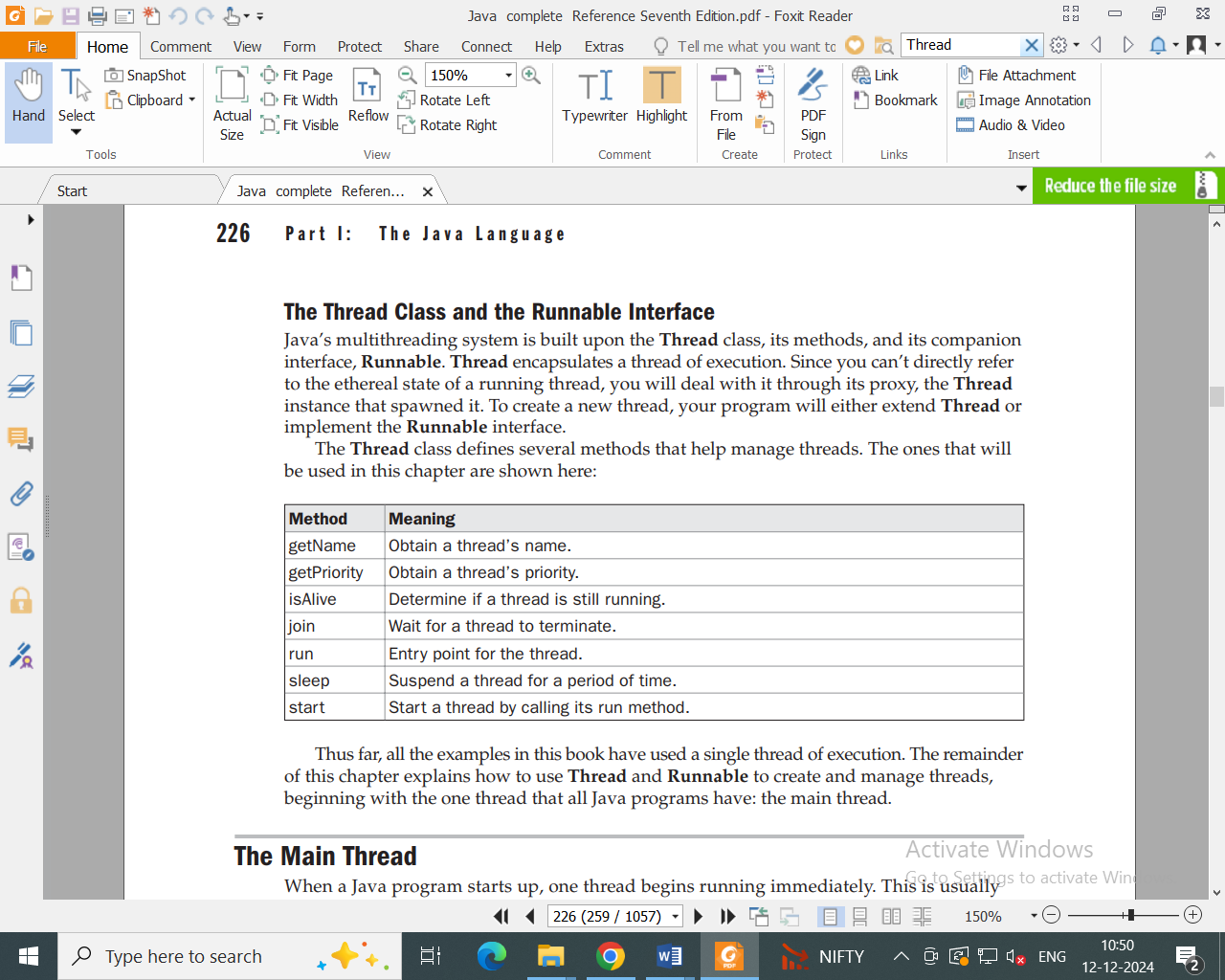
|  |  |
| --- | --- |
| **Process** | **Thread** |
| Executing several tasks simultaneously where each task is a separate  independent process such type of multitasking is called process based multitasking. | Executing several tasks simultaneously where each task is a separate independent part of the same program. Such type of  multitasking is called thread based multitasking. |
| Process is heavy weight component. | Thread is a light weight component. |
| Each process has a separate memory address | All threads share same memory address |
| Process is uncontrollable | Thread is controllable |
| IPC(Inter Process Communication) is expensive and limited | Inter Thread Communication(ITC) is inexpensive. |
| Context switching from one process to another process is also costly. | Context switching from one thread to another thread is low cost. |

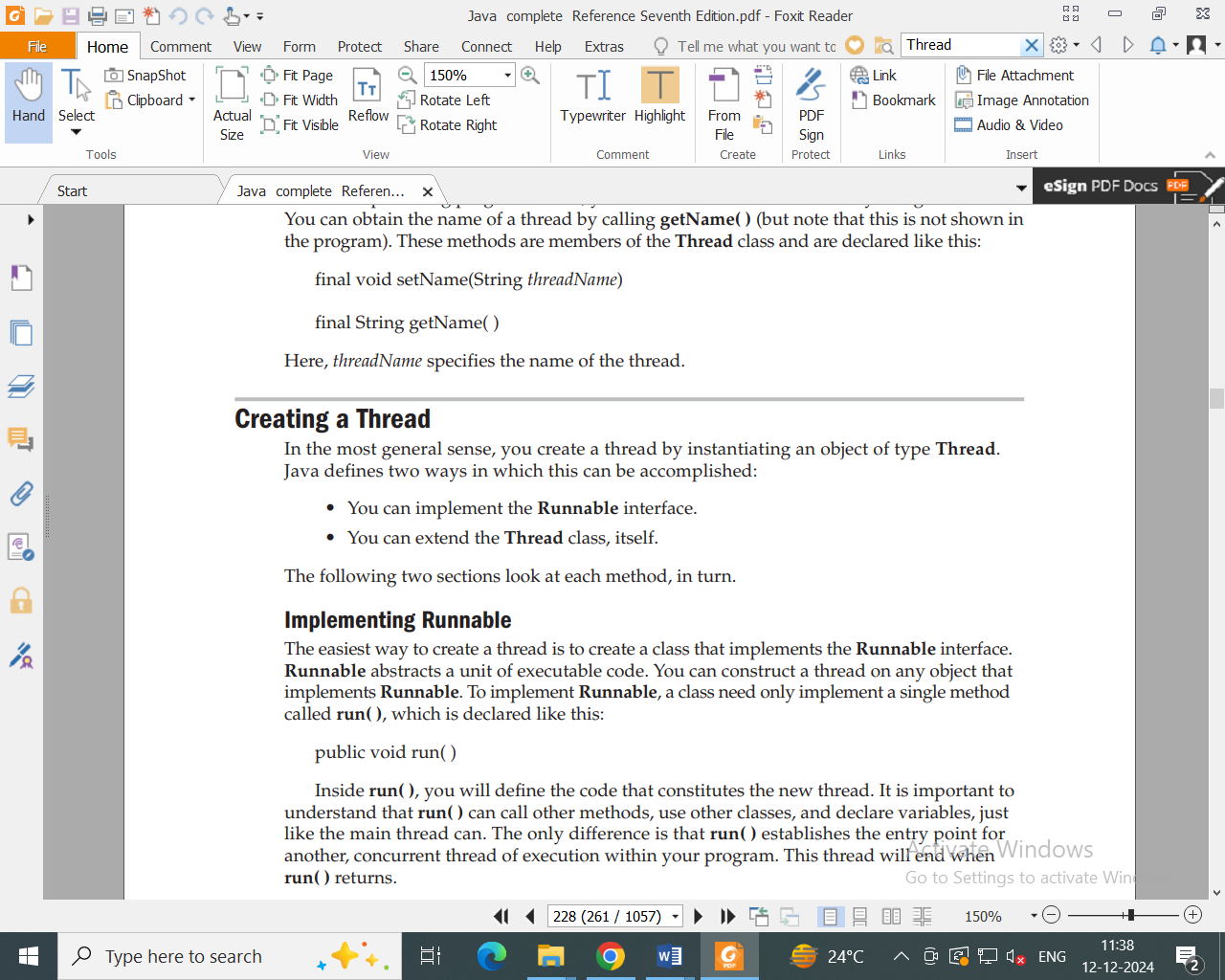
Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn’t run any faster than a lower- priority thread if it is the only thread running. Instead, a thread’s priority is used to decide when to switch from one running thread to the next. This is called a context switch. The rules that determine when a context switch takes place are simple:

1. *A thread can voluntarily relinquish control.* This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
2. *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking.*

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers.





**Method-1 (By implements Runnable interface)**

We can define a thread even by implementing Runnable interface directly.

Runnable interface present in java.lang package and contains only one method i. e run().

interface Runnable()

{

public void run();

}

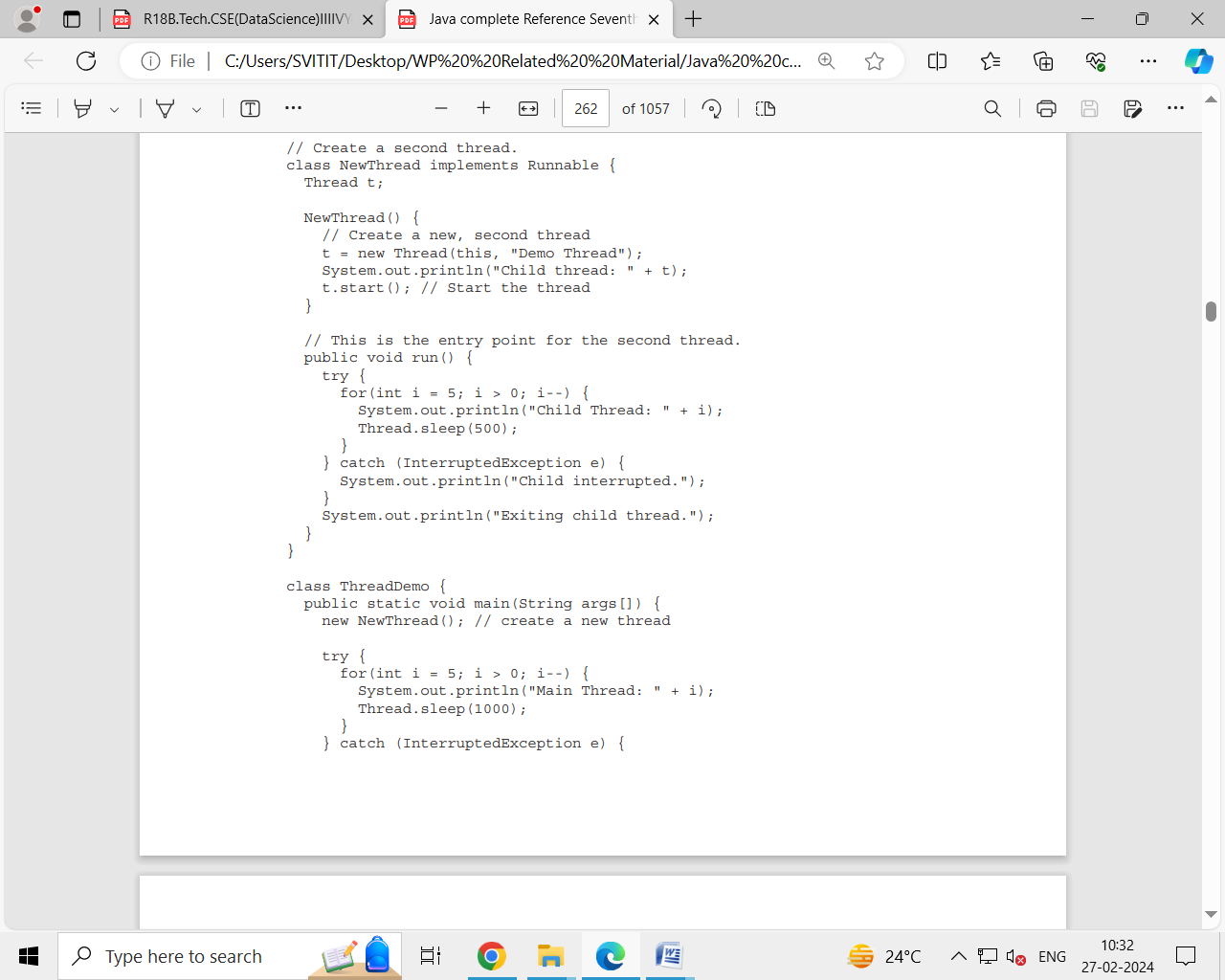
After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

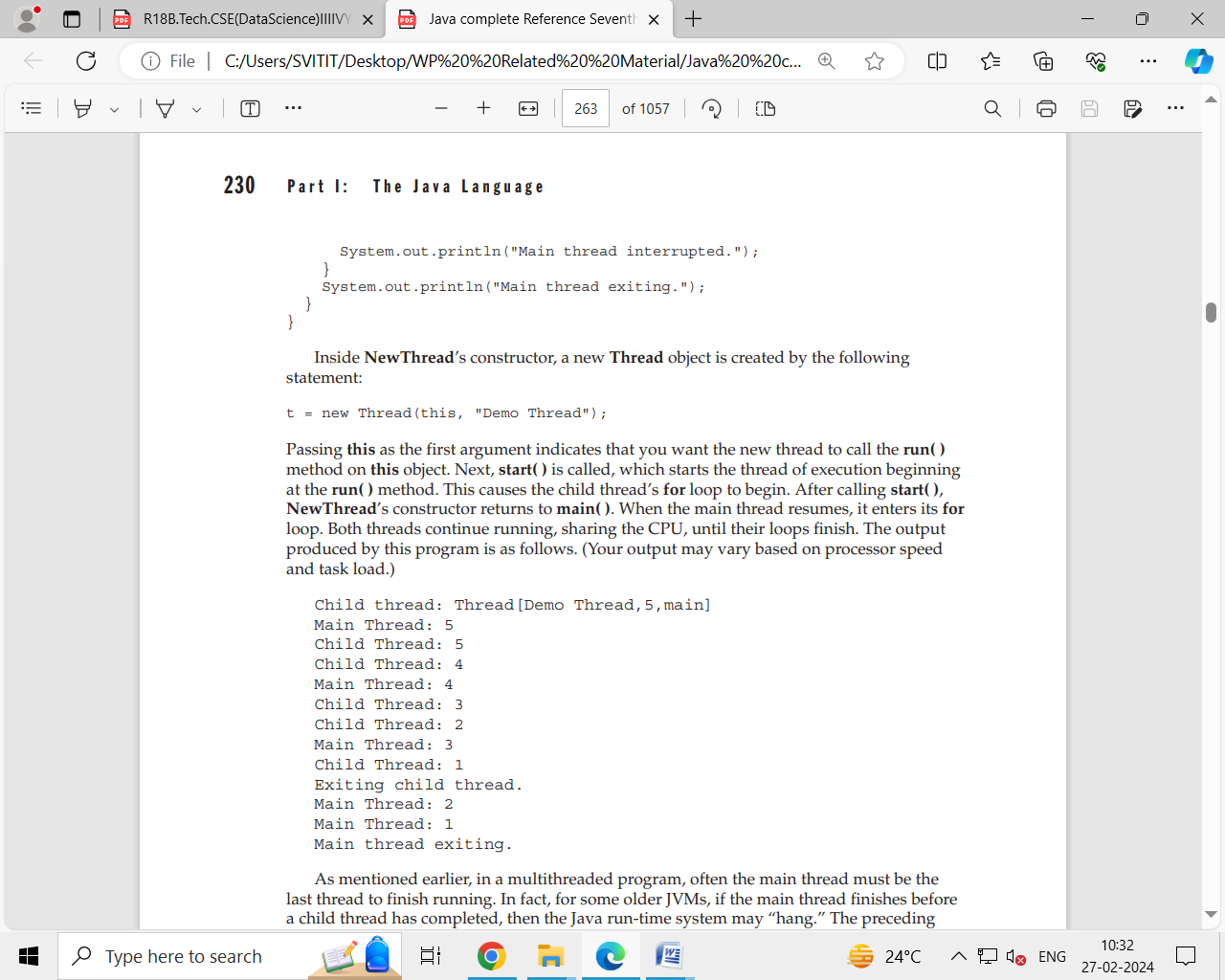
***Thread(Runnable threadOb, String threadName)***

In this constructor, threadOb is an instance of a class that implements the Runnable interface.This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ). The start( ) method is shown here:

**//demonstrating program.**





**Method: 2 ( by extends from Thread class)**

Public class mythread extends Thread

{

Public void run()

{

for(int i=0;i<10;i++)

System.out.println(“Child Thread”);

} }

Class ThreadDemo

{

Public static void main(string args[])

{

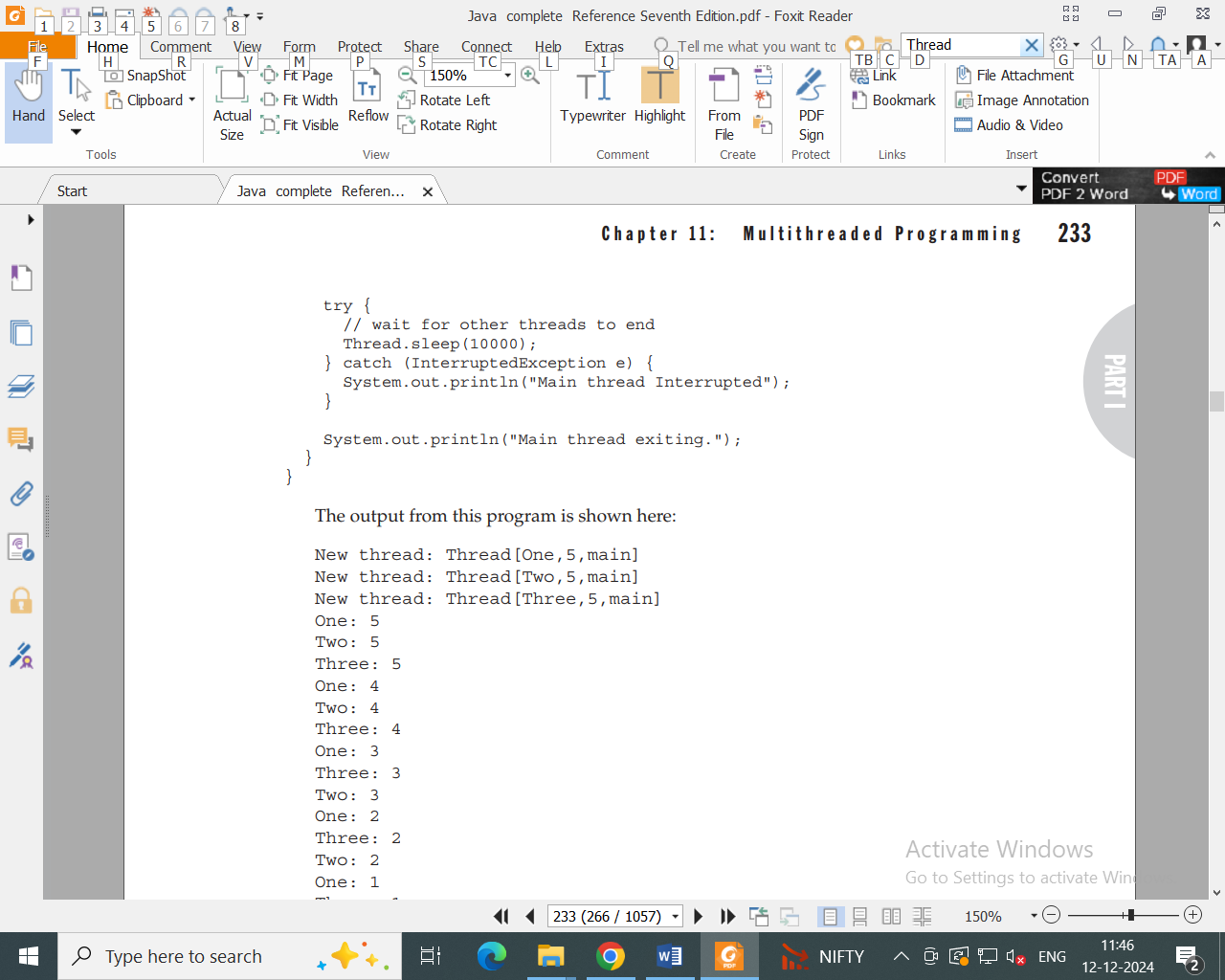
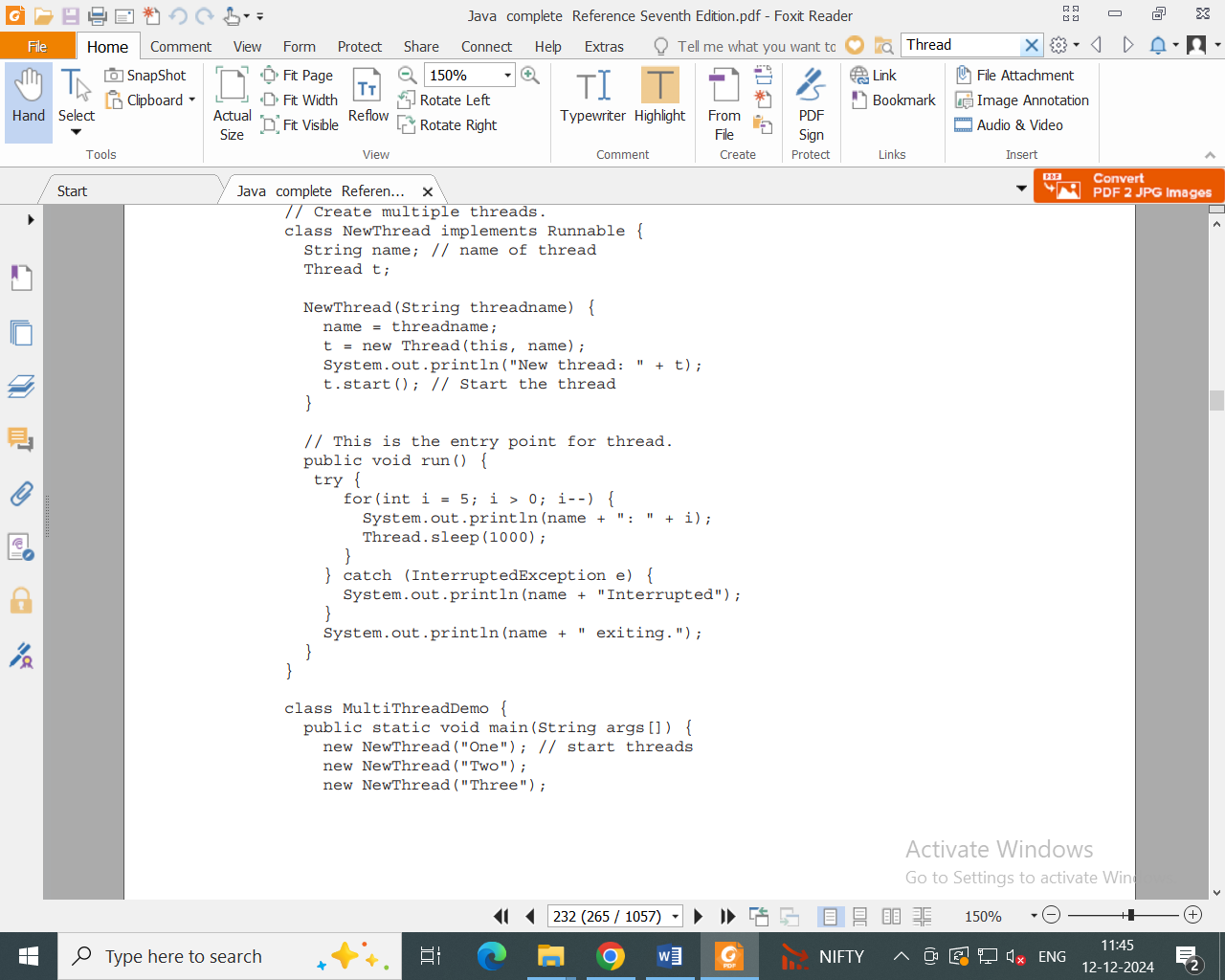
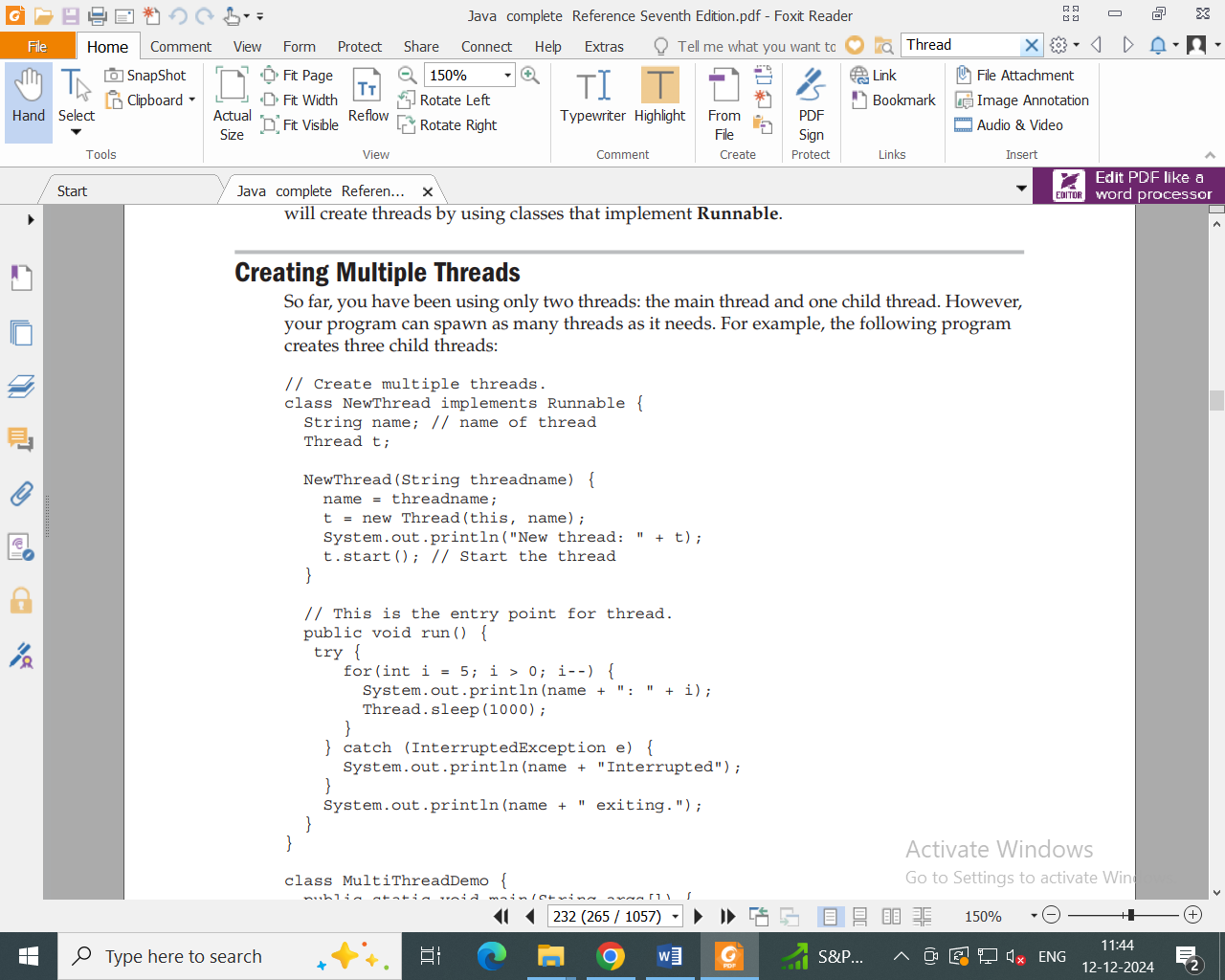
mythread t=new mythread();

t.start();

for(int i=0;i<10;i++)

System.out.println(“Parent Thread”);

} }



**Thread Priorities**

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower priority thread.

Every thread in java has some priority. The range of thread priorities is 1 to **10 (1—>least, 10—>highest)**.

Thread class defines the following constants to represent some standard priorities

Thread.MAX\_PRIORITY whose value is 10

Thread.NORM\_PRIORITY whose value is 5

Thread.MIN\_PRIORITY whose value is 1

The thread priorities used by thread Scheduler while allocating CPU. The thread which is having highest priority will get chance first for execution.

**Default Priority**

The default priority for only main thread is ‘5’. But for all the remaining threads it will be inherited from parent to child i.e whatever will be the parent thread has the priority the same will be the priority of child thread.

Thread class defines the following methods to get and set priority of a thread.

Public final int getPriority();

Public final int setPriority(int priority);

it should be from 1 to 10 otherwise we will get run time Exception saying **”illegalArgumentException”**

**Example**

class MyThread extends Thread

{

public void run()

{

for(int i=0;i<5;i++)

{

System.out.println ("Child Thread");

}}}

class ThreadPriorityDemo

{

public static void main(String args[])

{

MyThread t=new MyThread();

t.setPriority(10);

t.start();

for(int i=0;i<5;i++)

{

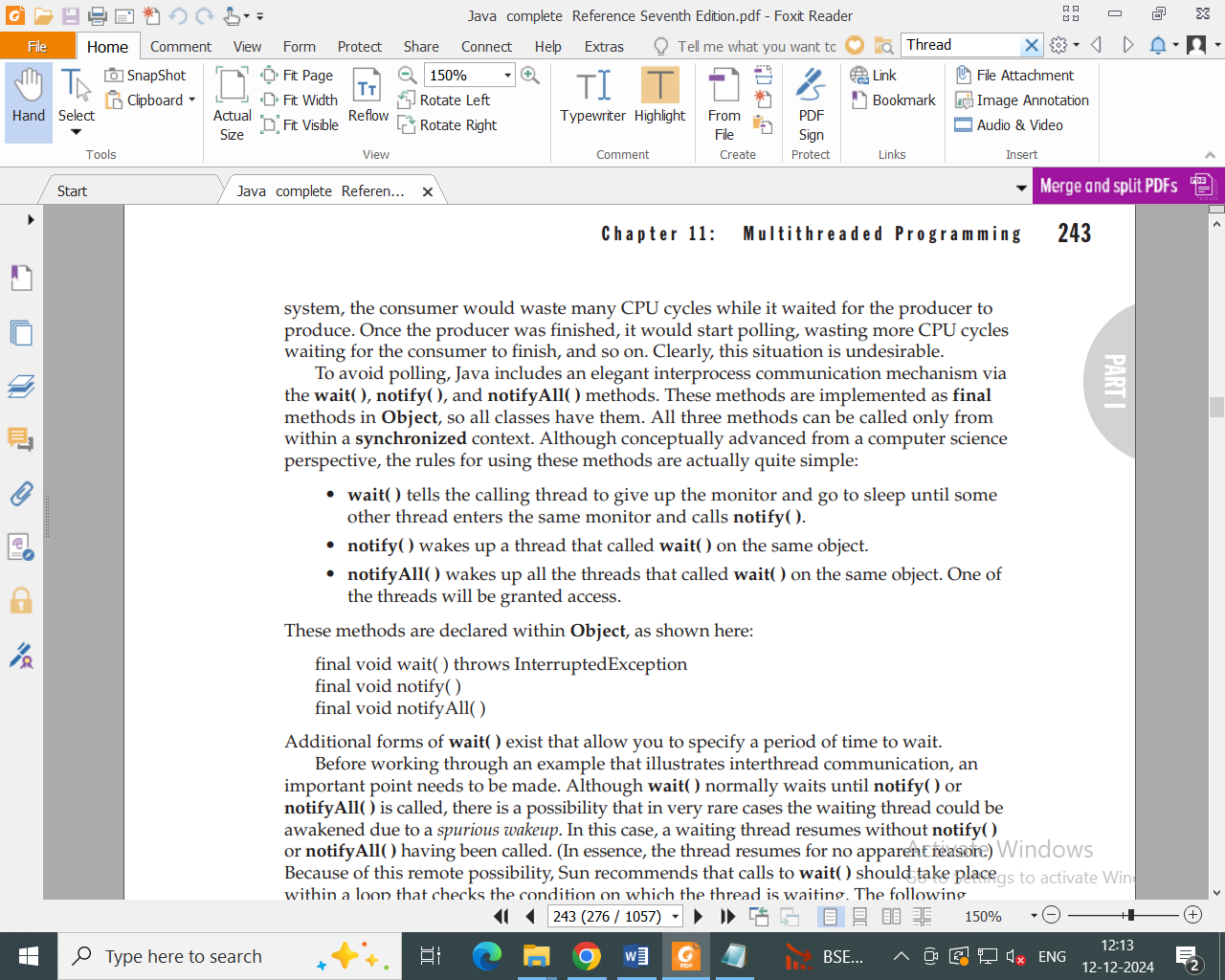
System.out.println ("Main Thread");

}}}

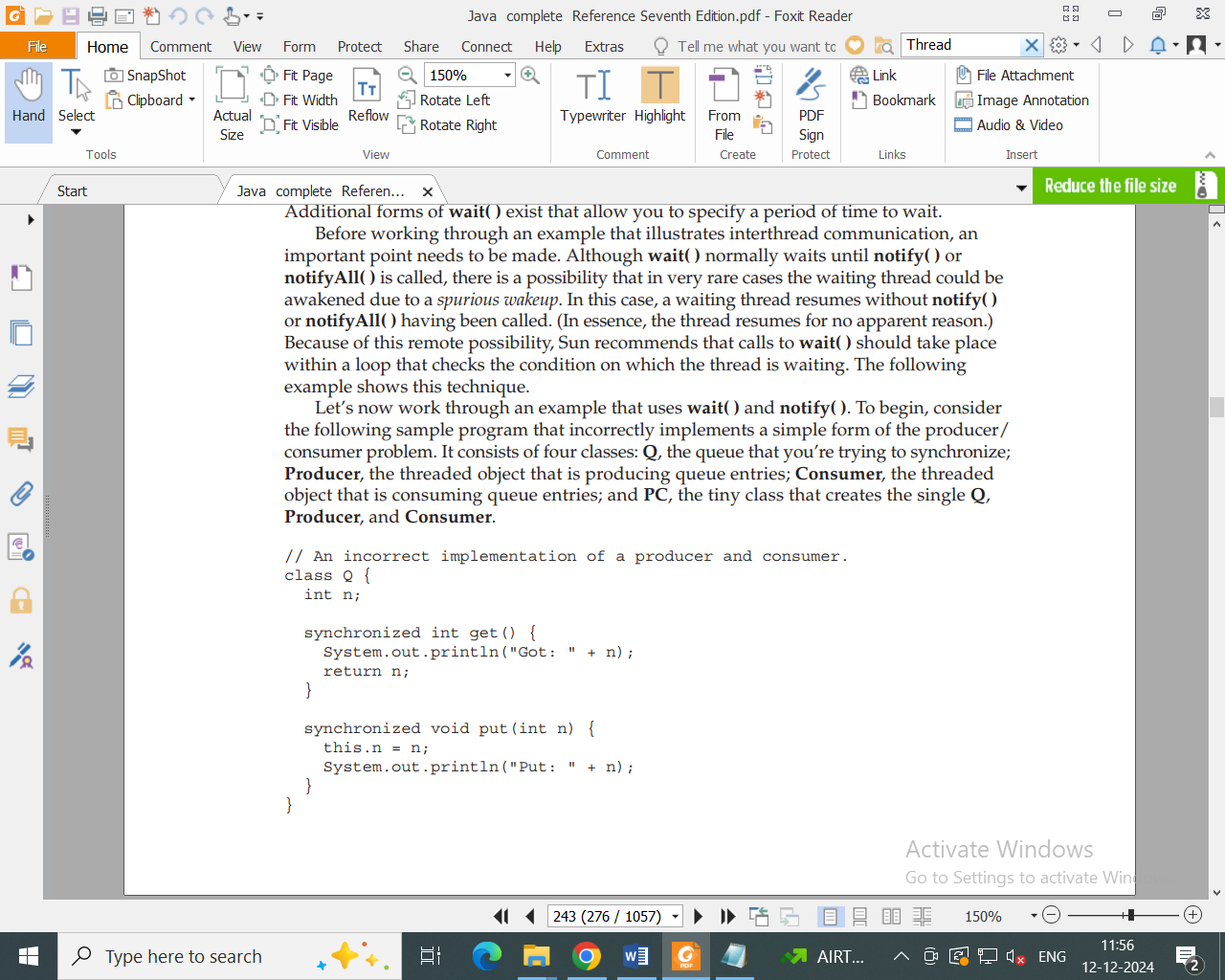
**Inter-thread Communication in Java**

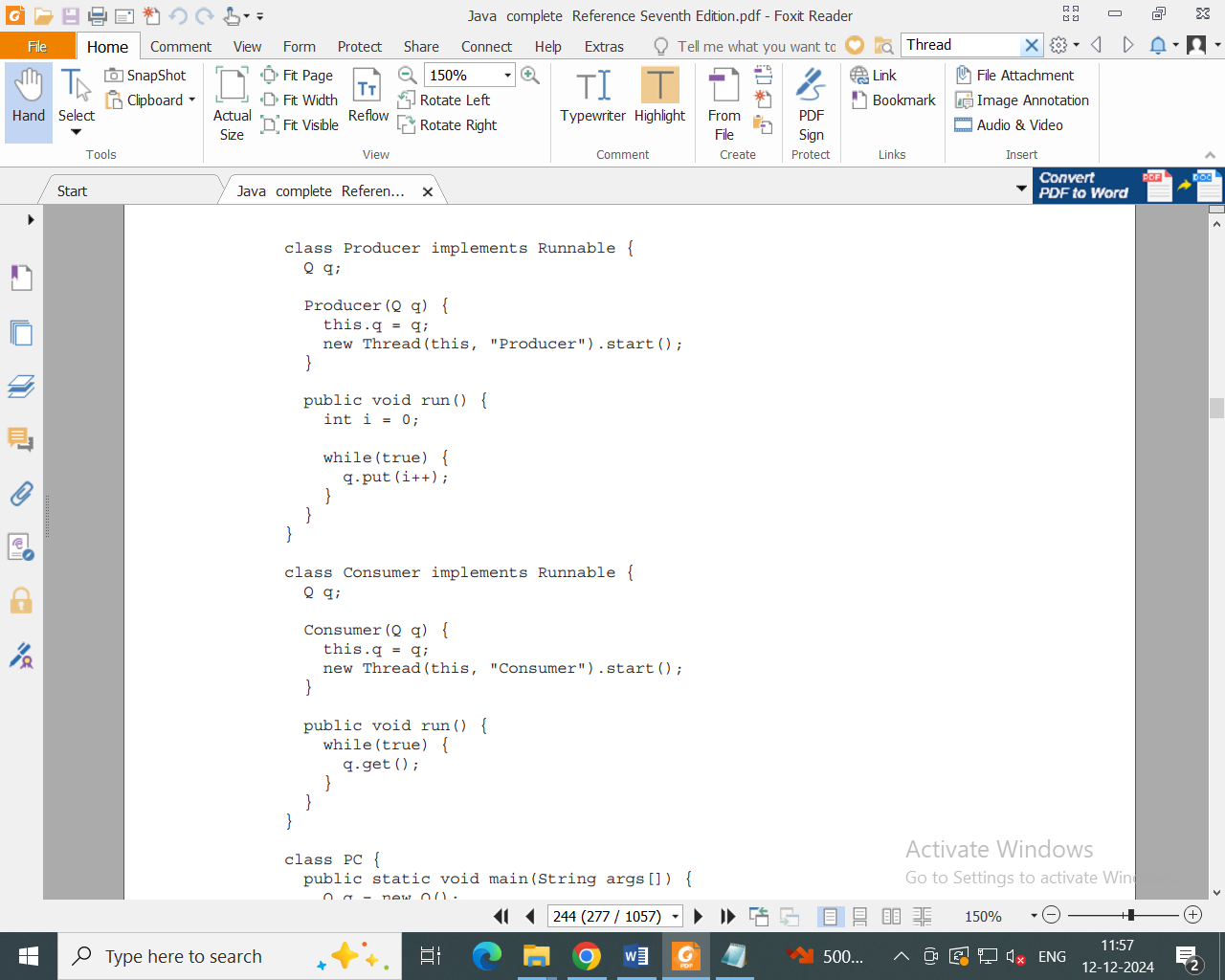
**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

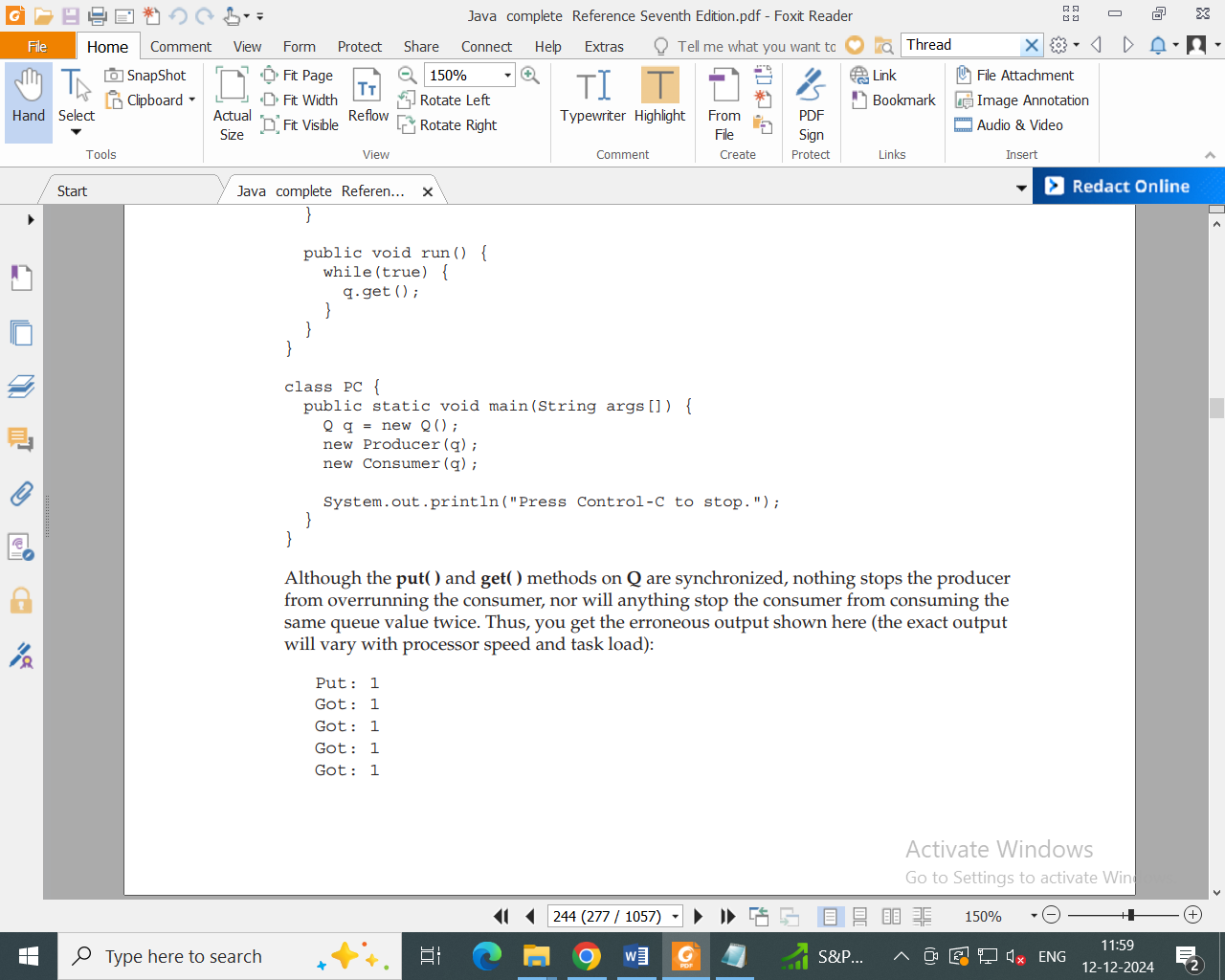
Multithreading replaces event loop programming by dividing your tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling

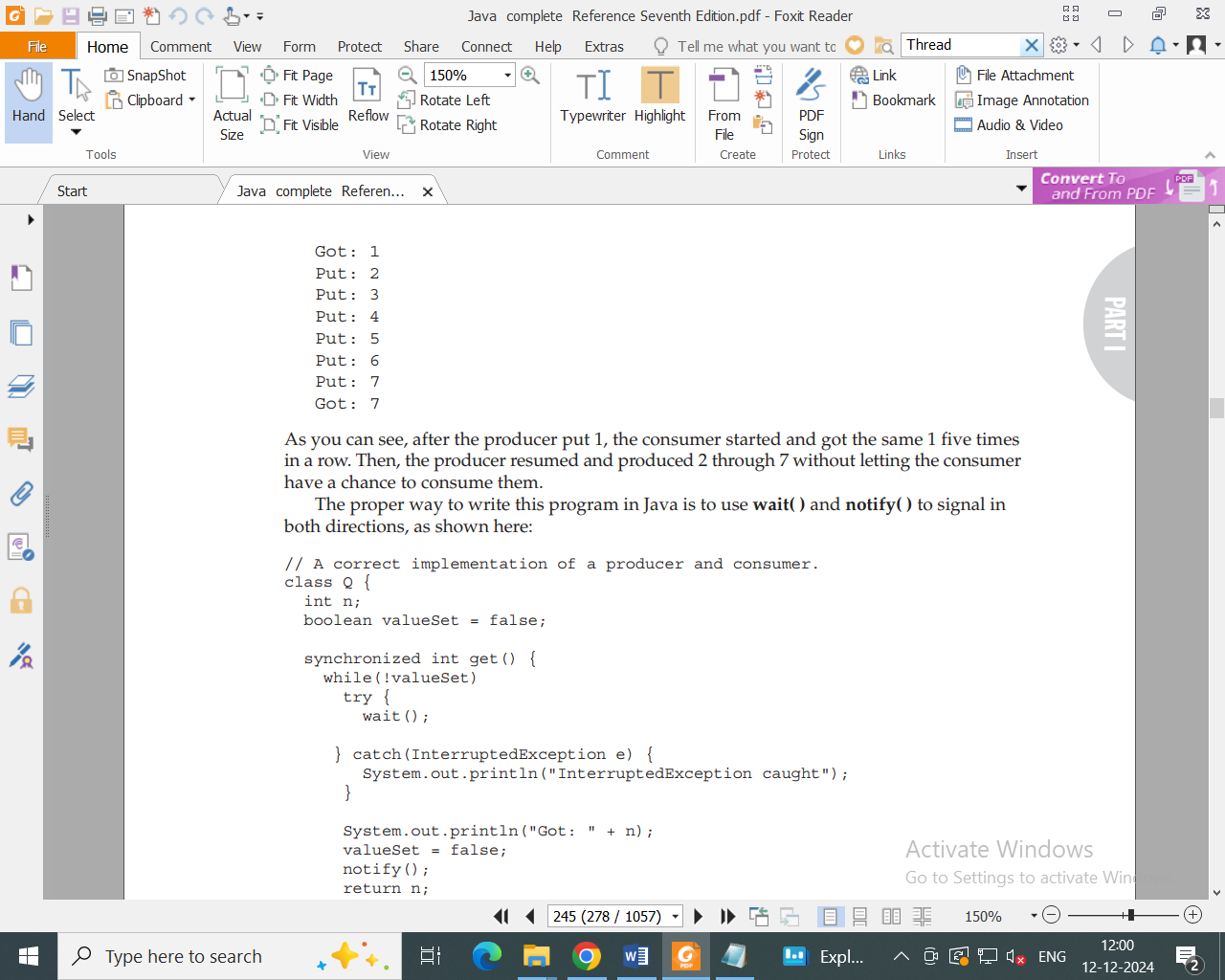


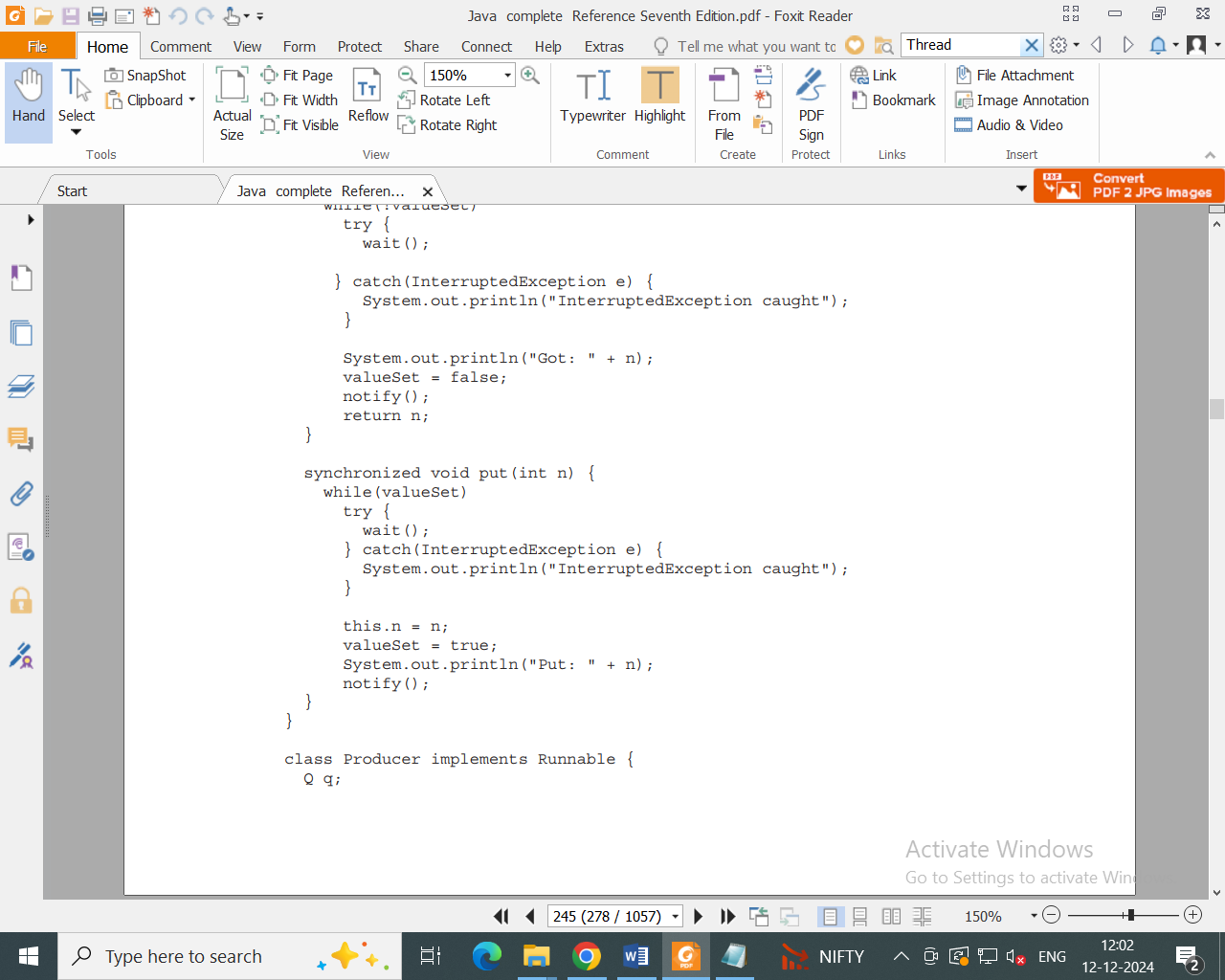
Let’s now work through an example that uses **wait( )** and **notify( )**. To begin, consider the following sample program that incorrectly implements a simple form of the producer/ consumer problem. It consists of four classes: **Q**, the queue that you’re trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

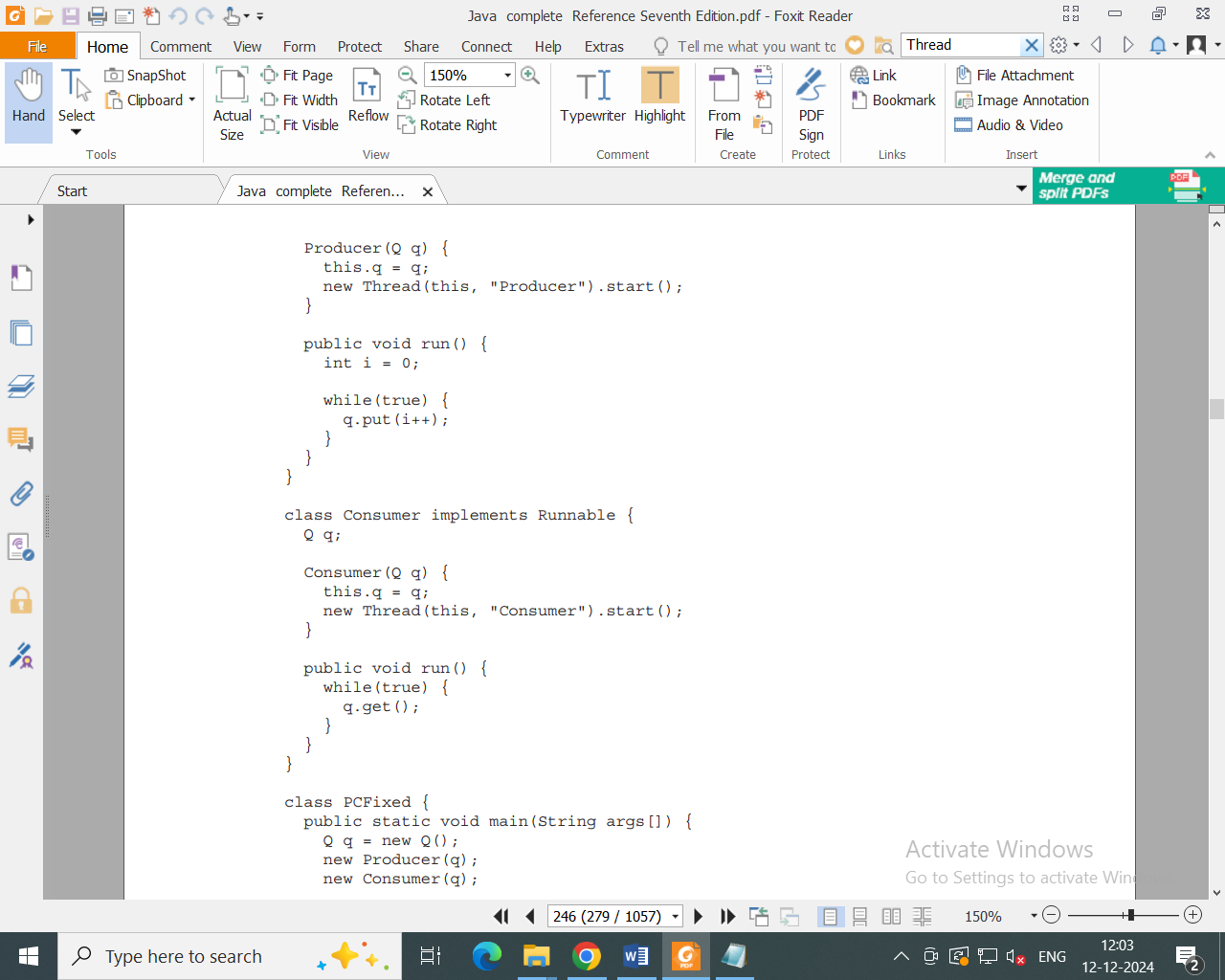


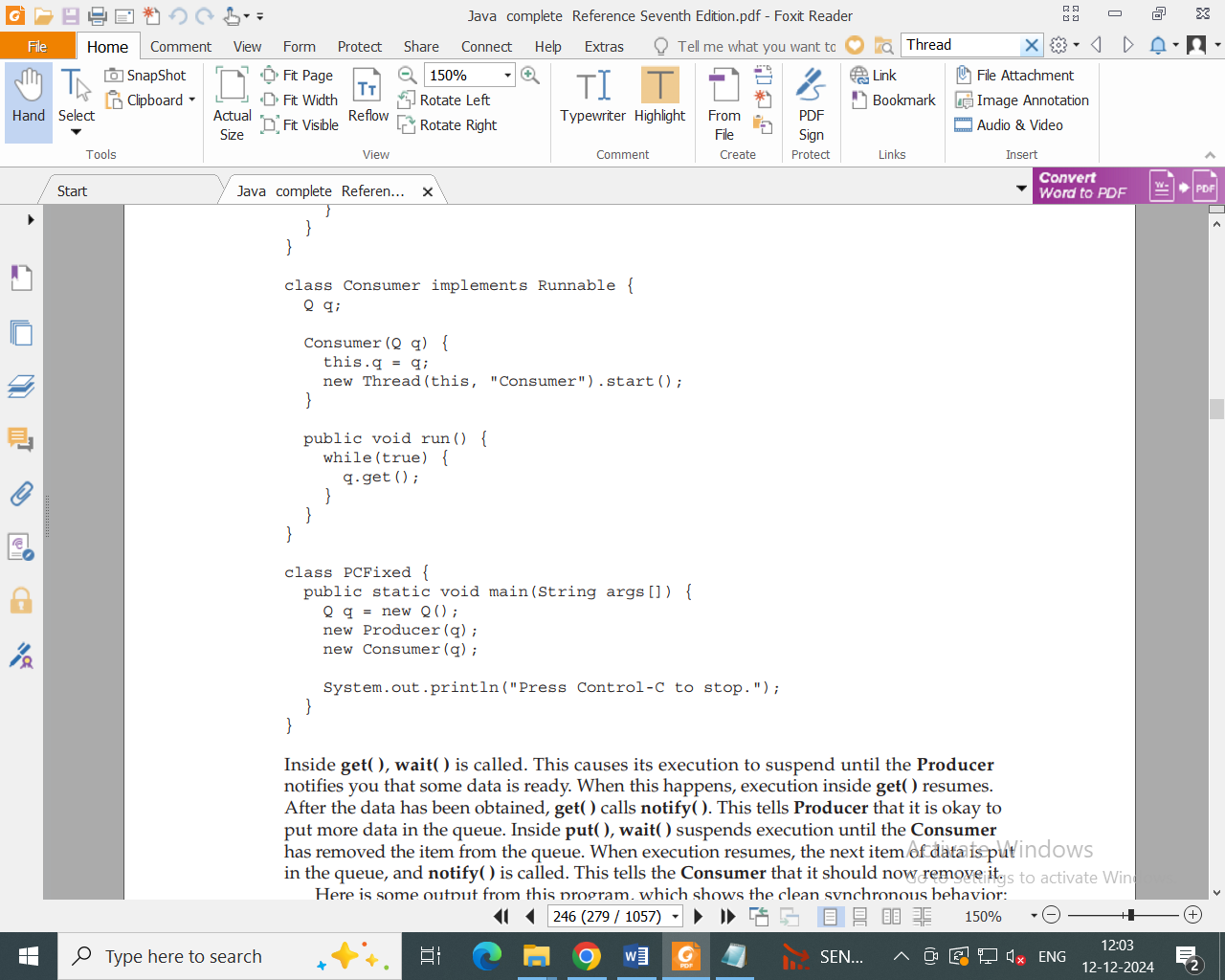


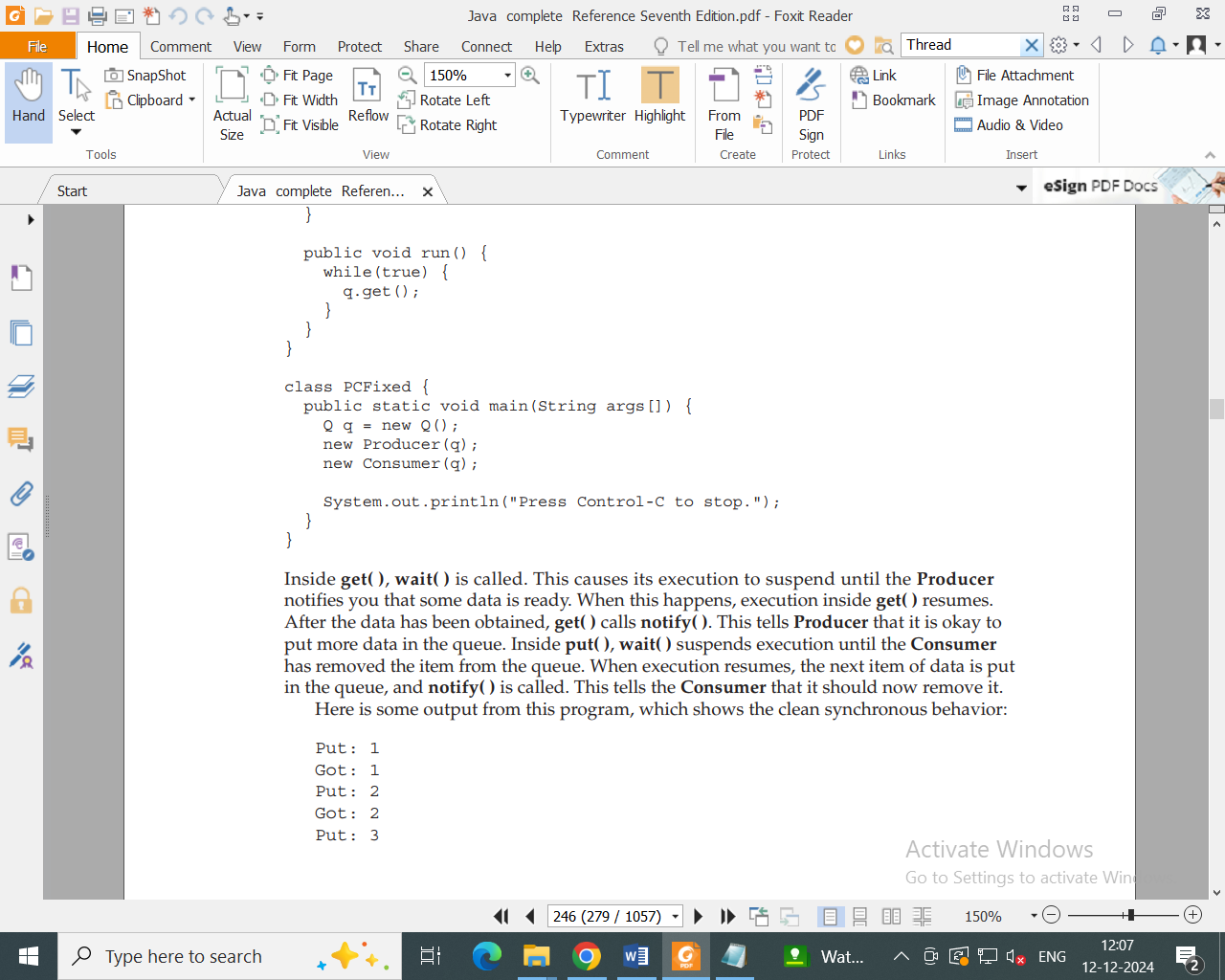


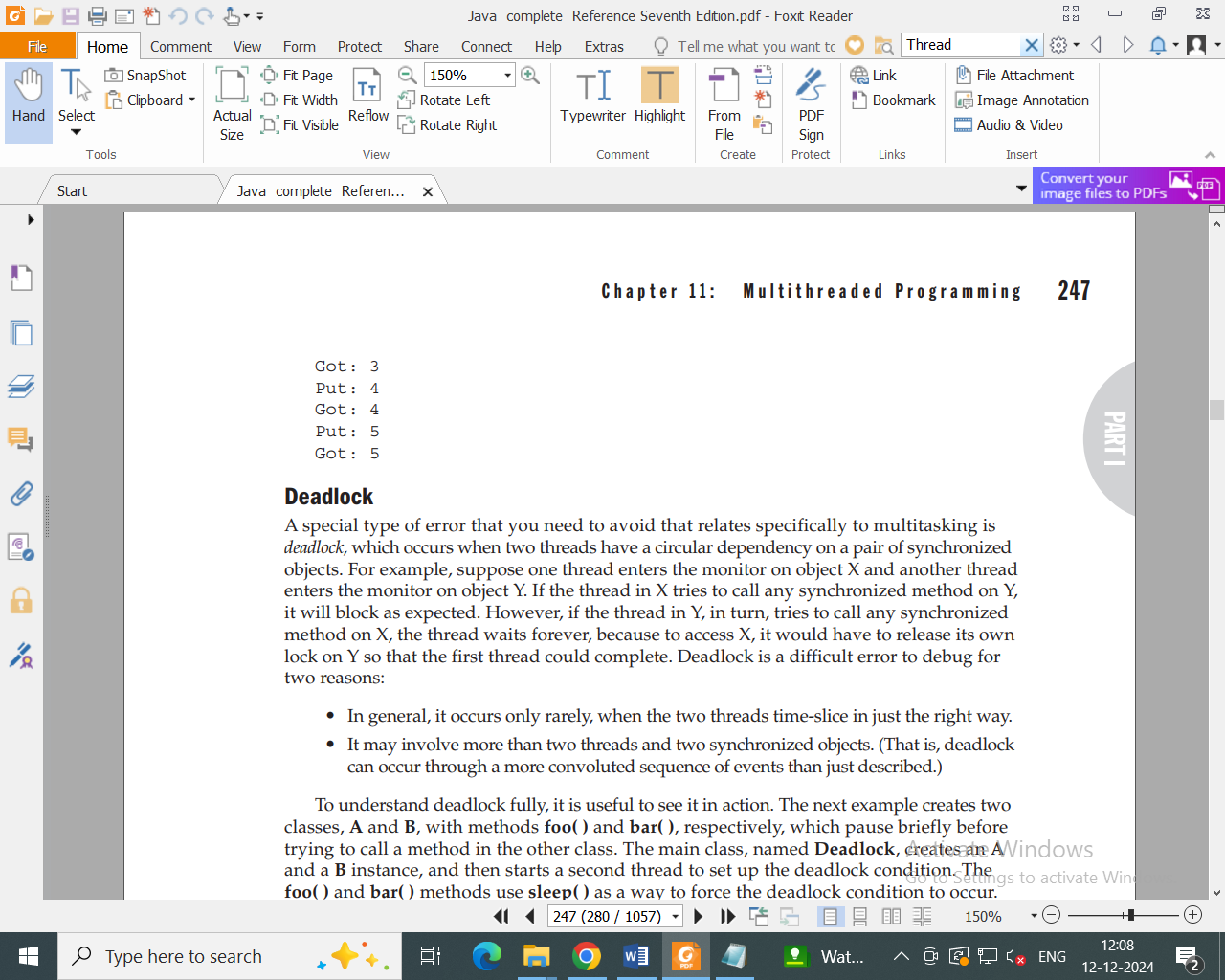












Notes on Daemon Thread, Synchronization is provided in class itself. Pls check notes.

Notes on enumneration,Autoboxing is provided in class. Pls check.