**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | 12 June 2020 | **Name:** | Rashmi KB |
| **Course:** | VLSI | **USN:** | 4AL16EC056 |
| **Topic:** | Basic CMOS inverter | **Semester & Section:** | 8th sem “B”section |
| **Github Repository:** | rashmikb |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| C:\Users\User\Downloads\WhatsApp Image 2020-06-12 at 8.12.15 PM (3).jpegC:\Users\User\Downloads\WhatsApp Image 2020-06-12 at 8.12.16 PM.jpeg  **C:\Users\User\Downloads\WhatsApp Image 2020-06-12 at 8.12.16 PM (1).jpegC:\Users\User\Downloads\WhatsApp Image 2020-06-12 at 8.12.16 PM (2).jpeg**  **C:\Users\User\Downloads\WhatsApp Image 2020-06-12 at 8.12.16 PM (3).jpeg**   |  | | --- | | Overview |   Complementary MOSFET (CMOS) technology is widely used today to form circuits in numerous and varied applications.  Today’s computers CPUs and cell phones make use of CMOS due to several key advantages.  CMOS offers low power dissipation, relatively high speed, high noise margins in both states, and will operate over a wide range of source and input voltages (provided the source voltage is fixed).  Next I will attempt to explain just how this logic gate works now that you have some idea of how important CMOS is in your day-to-day life.         |  | | --- | | CMOS Inverter Basics |   As you can see from Figure 1, a CMOS circuit is composed of two MOSFETs.  The top FET (MP) is a PMOS type device while the bottom FET (MN) is an NMOS type.  The body effect is not present in either device since the body of each device is directly connected to the device’s source.  Both gates are connected to the input line.  The output line connects to the drains of both FETs.    Take a look at the VTC in Figure 2.  The curve represents the output voltage taken from node 3.  You can easily see that the CMOS circuit functions as an inverter by noting that when VIN is five volts, VOUT is zero, and vice versa. Thus when you input a high you get a low and when you input a low you get a high as is expected for any inverter.  You might be wondering what happens in the middle, transition area of the curve.  You might also be curious as to what modes of operation the MOSFETs are in.  We will look at these issues next.       |  |  | | --- | --- | | https://courseware.ee.calpoly.edu/~dbraun/courses/ee307/F02/02_Shelley/cmos_files/image002.gif                            Figure:  CMOSInverter | https://courseware.ee.calpoly.edu/~dbraun/courses/ee307/F02/02_Shelley/cmos_files/image004.jpg  Figure :  Basic Voltage Transfer  Characteristic |          |  | | --- | | DC Analysis |   Figure 3 shows a more detailed VTC.  Before we begin our analysis it is important to mention three items.     * The MOSFETS must be perfectly matched for optimum operation, that is, they must have the same threshold voltage magnitude and conduction parameter. * The drain current (ID) through the NMOS device equals the drain current through the PMOS device at all times.  MOSFET gates have a high input impedance and we assume the circuit’s output sees no significant loading. * VDD equals the voltage across the PMOS plus the voltage across the NMOS by KVL.     https://courseware.ee.calpoly.edu/~dbraun/courses/ee307/F02/02_Shelley/cmos_files/image006.gif  **Figure :  VTC with Input Signal**   Static behaviour Often measurements can only be taken while a system is in a stable state (i.e. at equilibrium), because changes within the system are too fast to be measured and would invalidate the measured data. *Model*s of such systems - when based exclusively on the empirical data from measurements - are often deliberately constrained to the system's behavior at that stationary point and leave out the more *[dynamic](https://people.montefiore.uliege.be/bullinger/sysbioDE/projects/glossary/STATIC_BEHAVIOR.shtml" \l "dynamic)* aspects of a systems behavior. A typical example is that of a bioreactor, which is clearly designed to operate around an equilibrium point that ensures optimal conditions for the microorganisms in the reactor. The model needed to run the reactor during production can be constrained to the equilibrium of optimal growth conditions and leave out other aspects, like start-up or shutdown phases.  Behavior within a small area around such an equilibrium point is called *local* *[dynamic](https://people.montefiore.uliege.be/bullinger/sysbioDE/projects/glossary/STATIC_BEHAVIOR.shtml" \l "dynamic)* *behavior* and can be represented by a linearization (*[dynamic](https://people.montefiore.uliege.be/bullinger/sysbioDE/projects/glossary/STATIC_BEHAVIOR.shtml" \l "dynamic)*) of the original system. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | 12 June 2020 | **Name:** | Rashmi KB |
| **Course:** | Java Tutorial for Complete Beginners | **USN:** | 4AL16EC056 |
| **Topic:** | Programming core java   1. Generics and Wildcards 2. Anonymous Classes 3. Reading Files Using Scanner 4. Handling Exceptions 5. Multiple Exceptions 6. Runtime vs. Checked Exceptions 7. Abstract Classes 8. Reading Files With File Reader 9. Try-With-Resources 10. Creating and Writing Text Files | **Semester & Section:** | 8th sem “B”section |
| **Github Repository:** | rashmikb |  |  |
| **AFTERNOON SESSION DETAILS** | | | |
| C:\Users\User\Downloads\WhatsApp Image 2020-06-12 at 8.12.15 PM.jpegC:\Users\User\Downloads\WhatsApp Image 2020-06-12 at 8.12.15 PM (1).jpegC:\Users\User\Downloads\WhatsApp Image 2020-06-12 at 8.12.15 PM (2).jpegWildcards Consider the problem of writing a routine that prints out all the elements in a collection. Here's how you might write it in an older version of the language (i.e., a pre-5.0 release):  **void** printCollection(Collection c) {  Iterator i = c.iterator();  **for** (k = 0; k < c.size(); k++) {  System.out.println(i.next());  }  }  And here is a naive attempt at writing it using generics (and the new for loop syntax):  **void** printCollection(Collection<Object> c) {  **for** (Object e : c) {  System.out.println(e);  }  }  The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes Collection<Object>, which, as we've just demonstrated, is **not** a supertype of all kinds of collections!  So what **is** the supertype of all kinds of collections? It's written Collection<?> (pronounced "collection of unknown"), that is, a collection whose element type matches anything. It's called a **wildcard type** for obvious reasons. We can write:  **void** printCollection(Collection<?> c) {  **for** (Object e : c) {  System.out.println(e);  }  }  and now, we can call it with any type of collection. Notice that inside printCollection(), we can still read elements from c and give them type Object. This is always safe, since whatever the actual type of the collection, it does contain objects. It isn't safe to add arbitrary objects to it however:  Collection<?> c = new ArrayList<String>();  c.add(new Object()); // Compile time error  Since we don't know what the element type of c stands for, we cannot add objects to it. The add() method takes arguments of type E, the element type of the collection. When the actual type parameter is ?, it stands for some unknown type. Any parameter we pass to add would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is null, which is a member of every type.  On the other hand, given a List<?>, we **can** call get() and make use of the result. The result type is an unknown type, but we always know that it is an object. It is therefore safe to assign the result of get() to a variable of type Object or pass it as a parameter where the type Object is expected. Anonymous Classes Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.  This section covers the following topics:   * [Declaring Anonymous Classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html" \l "declaring-anonymous-classes) * [Syntax of Anonymous Classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html" \l "syntax-of-anonymous-classes) * [Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html" \l "accessing) * [Examples of Anonymous Classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html" \l "examples-of-anonymous-classes)  Declaring Anonymous Classes While local classes are class declarations, anonymous classes are expressions, which means that you define the class in another expression. The following example, [HelloWorldAnonymousClasses](https://docs.oracle.com/javase/tutorial/java/javaOO/examples/HelloWorldAnonymousClasses.java" \t "_blank), uses anonymous classes in the initialization statements of the local variables frenchGreeting and spanishGreeting, but uses a local class for the initialization of the variable englishGreeting:  public class HelloWorldAnonymousClasses {    interface HelloWorld {  public void greet();  public void greetSomeone(String someone);  }    public void sayHello() {    class EnglishGreeting implements HelloWorld {  String name = "world";  public void greet() {  greetSomeone("world");  }  public void greetSomeone(String someone) {  name = someone;  System.out.println("Hello " + name);  }  }    HelloWorld englishGreeting = new EnglishGreeting();    HelloWorld frenchGreeting = new HelloWorld() {  String name = "tout le monde";  public void greet() {  greetSomeone("tout le monde");  }  public void greetSomeone(String someone) {  name = someone;  System.out.println("Salut " + name);  }  };    HelloWorld spanishGreeting = new HelloWorld() {  String name = "mundo";  public void greet() {  greetSomeone("mundo");  }  public void greetSomeone(String someone) {  name = someone;  System.out.println("Hola, " + name);  }  };  englishGreeting.greet();  frenchGreeting.greetSomeone("Fred");  spanishGreeting.greet();  }  public static void main(String... args) {  HelloWorldAnonymousClasses myApp =  new HelloWorldAnonymousClasses();  myApp.sayHello();  }  } Checked and unchecked exceptions in java with examples There are two types of exceptions: checked exception and unchecked exception. In this guide, we will discuss them. The main **difference between checked and unchecked exception** is that the checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime. What are checked exceptions? Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using [try-catch block](https://beginnersbook.com/2013/04/try-catch-in-java/) or it should declare the exception using [throws keyword](https://beginnersbook.com/2013/04/java-throws/), otherwise the program will give a compilation error.  Lets understand this with the help of an **example**: Checked Exception Example In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where a checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException. | | | |