**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **12-06-2020** | **Name:** | **Kiran N** |
| **Course:** | **JAVA** | **USN:** | **4al16ec031** |
| **Topic:** | **Programming core java**  **Generics and Wildcards, Anonymous Classes**  **Reading Files Using Scanner, Handling**  **Exceptions, Multiple Exceptions, Runtime**  **vs. Checked Exceptions, Abstract Classes,**  **Reading Files With File Reader, Try-With-**  **Resources, Creating and Writing Text Files** | **Semester & Section:** | **8th and A** |
| **Github Repository:** | **Kiran-course** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Programming**      Wildcards  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  •Syntax of Anonymous Classes  •Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the  Anonymous Class  •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,  Hello World Anonymous Classes, 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);  }  }  HelloWorldenglishGreeting = new EnglishGreeting();  HelloWorldfrenchGreeting = 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);  }  };  HelloWorldspanishGreeting = 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) {  HelloWorldAnonymousClassesmyApp =  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 or it should declare the exception using throws keyword, otherwise the program will give a compilation error |