Module 11 - Classes and Interfaces

# Overview and Objectives

## OVERVIEW

In this module we will learn about classes and interfaces, the heart of Java and many other OO language and the basic units of abstraction. We will learn guidelines that can help us make usable, robust, and flexible classes and interfaces.

## COURSE LEVEL OBJECTIVES (CLO)

Upon completion of this course, you should be able to:

1. Construct modern high quality software systems and reason about them.
2. Properly define software specifications and rep-invariants.
3. Leverage immutability to properly construct threat safe programs.
4. Explain object-oriented concepts such as information hiding, encapsulation, data and type abstraction, and polymorphism.
5. Properly use exception handling
6. Identify when it is appropriate to use inheritance and generics.

## MODULE LEVEL OBJECTIVES (MLO)

Upon completion of this module’s activities, you should be able to:

1. explain and demonstrate important concepts about classes and interfaces (CLO 1, CL04, CLO6)
2. distinguish and proper use of composition and inheritence (CLO 1, CL04, CLO6)
3. distinguish and proper use of abstraction class and interface (CLO 1, CL04, CLO6)

# Module Video (Wiley-Produced w/Dan Ramos) [3-5 minutes]

# Learning Materials [~100 pages, ~3.5 hours]

## TEXTBOOK READINGS

* Joshua Bloch. Effective Java. Third Edition. Addison-Wesley Professional, 2017, ISBN 978-0-13-468599-1.
  + Chapter 4: Classes and Interfaces

# Learning Unit 1 – Classes and Interaces (MLO 1,2,3) [4 hours]

## Item 15: Minimize the accessibility of classes and members

* Information hiding is good:
  + It decouples the components of a system, it ensures that no other component is modifying the state of your components. This allows components to be **developed, tested, optimized, used, understood, and modified in isolation**
  + This speeds up system development because **components can be developed in parallel**
  + This **increases software reuse because components that are decoupled are often useful in other contexts**
  + If you make a class **public, you are obligated to support it forever** to maintain backward compatibility, i.e., **not break existing client code**
  + So, should make all members **private, but if need be, make them package-private**
    - Package-private: default, no access control keyword, part of the internal implementation
    - **private** and **package-private** members are not visible to the external clients
* **Why not make them protected?**
  + A **protected** member is part of the class's exported API and must be **supported forever** (subclassing).
  + Also, protected members could be made publically accessible via subclassing.
* If a method overrides a superclass method, it cannot have a more restrictive access level in the subclass than in the superclass… prevented by the compiler
  + Client code is written in the context of the superclass, you cannot prevent method access later on
* This is necessary to ensure that an instance of the subclass is usable anywhere that an instance of the superclass is usable (the Liskov substitution principle):
  + if A.foo() is public, B.foo() cannot be private.
* The above is applicable to method overriding. Is below allowed?
* class A{  
   public int x;  
  }  
  class B extends A {  
   private int x;  
  }  
  + Yes, these are different variables (no overriding!)
* To facilitate testing your code, it might be acceptable to change a **private member** to **package-private** to test it, but no higher
  + So tests can be made to be part of the package being tested
* **Why making instance fields public is bad?**
  + You should never change the **name and type** of a public field
  + Public mutable fields are bad for **thread-safety**
  + If an instance field is non-final or is a reference to a mutable object, then by making it public, **you give up the ability to enforce invariants** involving the field
* **static fields of public classes could be made public**, but must be **final** and contain either primitive values (immutables) or references to immutable objects
* Is it OK to make a static field public + final + references an array?
* public static final Thing[] VALUES = { ... }; // What could go wrong?
  + VALUES is final; entries in VALUES are not!
* To fix the problem, you can make the public array private and add a public immutable list:
* private static final Thing[] PRIVATE\_VALUES = { ... };  
  public static final List<Thing> VALUES = Collections.unmodifiableList(Arrays.asList(PRIVATE\_VALUES));
* Alternatively, you can make the array private and add a public method that returns a clone of it:
* private static final Thing[] PRIVATE\_VALUES = { ... };  
  public static final Thing[] values() {  
   return PRIVATE\_VALUES.clone();  
  }

## Item 16: In public classes, use accessor methods, not public fields

* Avoid code such as: class Point { public double x; public double y; }
  + Why? See item 15
* Also, public mutable fields are **not** thread safe
* Use get/set methods instead: public double getX() { return x; } public void setX(double x) { this.x = x}
* Advice holds for immutable fields as well. But why? They are thread-safe…
  + Limits possible ways for class to evolve
* **It is never OK to have public fields in public classes:**
  + if a class is public, make the fields private or package-private then provide accessor methods; this preserves the flexibility to change the class’s internal representation
  + **If a class is package-private**, there is nothing wrong with exposing its data fields. The client could never see it. Also, If a method is declared with default access, it can only be overridden by methods in the same package

## Item 18: Favor composition over inheritance

* When is it is safe to use inheritance?
  + Within a package, where the subclass and the superclass implementations are under the control of the same programmers
  + When extending classes are specifically designed and documented for extension, e.g., interface inheritance
  + Recall adding significant attributes might mess up the equality contract!!!
* Inheritance violates encapsulation:
  + A subclass depends on the implementation details of its superclass
  + A subclass might need to know the implementation of the superclass (see example next)
  + If the superclass changes, the subclass may break, even though its code has not been touched
  + Therefore, a subclass must evolve in tandem with its superclass, unless the superclass’s authors have designed it specifically for the purpose of being extended
  + Difficult for superclass to maintain invariants in face of malicious/careless subclass (mutability lecture)
* Suppose we want to track how many elements have been added to a HashSet, so we extend HashSet and override add() and addAll():

public class InstrumentedHashSet<E> extends HashSet<E> {  
 // The number of attempted element insertions  
 private int addCount = 0;  
 public InstrumentedHashSet() { }  
 public InstrumentedHashSet(int initCap, float loadFactor) {  
 super(initCap, loadFactor);  
 }  
 @Override public boolean add(E e) {  
 addCount++;  
 return super.add(e);  
 }  
 @Override public boolean addAll(Collection<? extends E> c) {  
 addCount += c.size();  
 return super.addAll(c);  
 }  
 public int getAddCount() { return addCount; }  
}

* This class looks reasonable, but it **doesn’t work**!!
* Suppose we create an instance and add three elements using the addAll method:
* InstrumentedHashSet<String> s = new InstrumentedHashSet<>();  
  s.addAll(Arrays.asList("Snap", "Crackle", "Pop"));
  + We would expect the getAddCount() method to return 3, but it returns 6!! What went wrong?
    - HashSet.addAll() is implemented on top of its add() method, which is reasonable…
    - InstrumentedHashSet.addAll() added 3 to addCount and then invokes HashSet.addAll() using super.addAll(). This in turn invokes the add() method, as overridden in InstrumentedHashSet, once for each element. Each of these three invocations added one more to addCount, for a total increase of 6
  + **Inheritance breaks encapsulation: overriding revealed how the superclass is implemented**
* **Fix1**: We could “fix” the subclass by not overriding HashSet.addAll().
  + However, this depends on the fact that HashSet.addAll() is implemented on top of its add() method, which not guaranteed to hold in all implementations of the Java platform.
* **Fix2**: We could override addAll() and call add() for each element.
  + This way HashSet.addAll() would no longer be invoked.
  + This amounts to reimplementing a superclass method (not good)
* **Fix3**: Composition: solves the problem even though it requires more work for the developer

// Note that an InstrumentedSet IS-A Set COMPOSITION  
public class InstrumentedSet<E> implements Set<E> {  
 private final Set<E> s;  
 private int addCount = 0;  
  
 public InstrumentedSet (Set<E> s) { this.s = s}  
  
 public boolean add(E o) {   
 addCount++; return s.add(o); // s is a Set, not InstrumentedSet  
 }  
  
 public boolean addAll (Collection<? extends E> c) {  
 addCount += c.size();  
 return s.addAll(c); // s is a Set, not InstrumentedSet  
 }  
 // forwarded methods from Set interface  
}  
 A more elegant solution next…

* **Fix4**: COMPOSITION followed by INHERITANCE

public class ForwardingSet<E> implements Set<E> {  
 private final Set<E> s; // Composition  
 public ForwardingSet(Set<E> s) { this.s = s; }  
 public void clear() { s.clear(); }  
 public boolean contains(Object o) { return s.contains(o); }  
 public boolean isEmpty() { return s.isEmpty(); }  
 public int size() { return s.size(); }  
 public Iterator<E> iterator() { return s.iterator(); }  
 public boolean add(E e) { return s.add(e); }  
 public boolean remove(Object o) { return s.remove(o); }  
 public boolean containsAll(Collection<?> c) {   
 return s.containsAll(c); }  
 public boolean addAll(Collection<? extends E> c) {   
 return s.addAll(c); }  
 public boolean removeAll(Collection<?> c) {   
 return s.removeAll(c); }  
 public boolean retainAll(Collection<?> c) {   
 return s.retainAll(c); }  
 public Object[] toArray() { return s.toArray(); }  
 public <T> T[] toArray(T[] a) { return s.toArray(a); }  
 …  
  
@Override public boolean equals(Object o) {   
 return s.equals(o); }  
@Override public int hashCode() { return s.hashCode(); }  
@Override public String toString() { return s.toString(); }  
}  
  
// Inheritance  
public class InstrumentedSet<E> extends ForwardingSet<E> {  
 //private final Set<E> s;  
 private int addCount = 0;  
 public InstrumentedSet(Set<E> s) { super(s); } // this.s = s  
 @Override public boolean add(E e) {  
 addCount++;  
 return super.add(e); // return s.add(e);   
 }  
 @Override public boolean addAll(Collection<? extends E> c)   
 {  
 addCount += c.size();  
 return super.addAll(c); // return s.addAll(c)  
 }  
 public int getAddCount() { return addCount; }  
}

* The InstrumentedSet class can be used to temporarily instrument a set instance that has already been used without instrumentation:
* static void walk(Set<Dog> dogs) {  
   InstrumentedSet<Dog> iDogs = new InstrumentedSet<>(dogs);  
   // Within this method use iDogs instead of dogs  
   // all changes (add/remove/…) are reflected in dogs!  
   // addCount is only updated when iDogs is used  
  }
* InstrumentedSet is as flexible as InstrumentedSet<E>, it supports HashSet, TreeSet and others:

Set<Instant> times = new InstrumentedSet<>(new TreeSet<> (comp));  
Set<E> s = new InstrumentedSet<>(new HashSet<> (INIT\_CAPACITY));

## Item 20: Prefer interfaces to abstract classes

* Both interfaces and abstract classes allow defining new types
* Both interfaces (since Java 8) and abstract classes allow providing default method implementations
* Due to single inheritance, interfaces are more flexible than abstract classes:
  + **Existing classes can easily be retrofitted to implement an additional new interface**: just specify implements and add the required methods.
    - Not so easy in the case of abstract classes (abstract1 must derive from abstract2, and abstract2 from abstract3…)
  + **Interfaces are ideal for defining mixin types**.
    - Types that a class can implement in addition to its “primary type”.
    - For example: 1) Comparable is a mixin interface that allows Automobile to declare that its instances are ordered; 2) Serializable is also mixin interface that allows Automobile to declare that its instances could be serialized
* Suppose we have an interface representing a singer and another representing a songwriter:
* public interface Singer { AudioClip sing(Song s); }   
  public interface Songwriter { Song compose(int chartPosition); }
* Some singers are also songwriters. So a single class can implement both Singer and Songwriter.
* In addition, we can define a third interface that extends both Singer and Songwriter:
* public interface SingerSongwriter extends Singer, Songwriter {  
   AudioClip strum();  
   void actSensitive();  
  }

### Abstract class vs Interface

* **Type of methods**: Abstract class can have abstract and non-abstract methods. Starting with Java 8, interfaces can have default and static methods.
* **Final Variables**: Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
* **Type of variables**: Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
* **Accessibility of Data Members**: Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

## Item 21: Design interfaces with future use in mind

* Prior to Java 8, if you added a new method to an interface, existing implementations would not compile
* In Java 8, the default method construct was added, which fixed that problem (adding a new default method will not break existing implementations)
* However, **adding new default methods to existing interfaces is risky**:
  + They could be used by all classes that implement the interface without the knowledge of the developer
  + The client could have access to functionality/data not intended to be made available

## Item 22: Use interfaces only to define types

* Some interfaces consists solely of static final fields, each exporting a constant
* Classes using these constants implement the interface to avoid the need to qualify constant names with a class name:
* public interface PhysicalConstants {   
   static final double AVOGADROS\_NUMBER = 6.022140857e23; // Boltzmann constant (J/K)  
   static final double BOLTZMANN\_CONSTANT = 1.38064852e-23; // Mass of the electron (kg)  
   static final double ELECTRON\_MASS = 9.10938356e-31;  
  }  
  public class ClassA implements PhysicalConstants {… }
* ClassA implementing PhysicalConstants:
  + Does not say anything about the type of ClassA (what functionality it provides)
  + "is-a" relationship?
  + Clients of a class that implements this interface don’t care about these constants!
  + It represents a commitment: if in a future release ClassA is modified so that it no longer needs to implement PhysicalConstants, it still must implement the interface to ensure binary compatibility (i.e., all code using ClassA will need to be recompiled if the "implements" is removed)
  + If a non-final class implements a constant interface, all of its subclasses will have access to the constants (unnecessary pollution)
* Better alternatives:
  + If the constants are strongly tied to an existing class, you should add them to the class; e.g., all of the boxed numerical primitive classes, such as Integer and Double, export MINVALUE and MAXVALUE constants
  + Otherwise, you should export the constants with a non-instantiable utility class, e.g.:
  + public class PhysicalConstants {  
     private PhysicalConstants() { } // Prevents instantiation  
     public static final double AVOGADROS\_NUMBER = 6.022140857e23;  
     public static final double BOLTZMANN\_CONST = 1.38064852e-23;  
     public static final double ELECTRON\_MASS = 9.10938356e-31;  
    }
* To access the constants, instead of fully qualifying them (e.g. PhysicalConstants.AVOGADROSNUMBER), you can make use of the static import facility:

import static PhysicalConstants.\*; // imports all “public static” fields and methods  
public class Test {  
 double atoms(double mols) { return AVOGADROS\_NUMBER \* mols; }  
 ...  
}

## Item 23: Prefer class hierarchies to tagged classes

* Tagged class: **a class whose instances come in several flavors and contain a tag field indicating the flavor**
* class Figure {   
  enum Shape { RECTANGLE, CIRCLE };  
  final Shape shape; // Tag field - the shape of this figure   
  double length; // These fields are used only if shape is RECTANGLE  
  double width;  
  double radius; // This field is used only if shape is CIRCLE  
  Figure(double radius) { // Constructor for circle  
   shape = Shape.CIRCLE; this.radius = radius;  
  }   
  Figure(double length, double width) { // Constructor for rectangle  
   shape = Shape.RECTANGLE; this.length = length; this.width = width;  
  }  
  double area() {  
   switch(shape) { // do your own dispatching!!  
   case RECTANGLE:  
   return length \* width;  
   case CIRCLE:  
   return Math.PI \* (radius \* radius);  
   default:  
   throw new AssertionError(shape);  
   }}}
* Tagged classes are bad:
  + Bad readability because multiple implementations are mixed together in a single class
  + To add a new flavor, you need to modify the source file: you must remember to add a case to every switch statement, or the class will fail at runtime
  + The data type of an instance gives no clue as to its flavor
* Subtyping is a better alternative
  + Define an abstract class. In our case it contains: “double area()”
  + If there are any methods/fields that do not depend on the value of the tag, put them in this class
  + Define concrete subclasses for each flavor: Circle and Rectangle
  + Include in each subclass the data fields particular to its flavor
  + Include in each subclass the appropriate implementation of each abstract method in the root class

abstract class Figure { // Class hierarchy replacement for a tagged class  
 abstract double area();  
}  
class Circle extends Figure {  
 double radius;  
 Circle(double radius) { this.radius = radius; }  
 @Override double area() { return Math.PI \* (radius \*radius); }  
}   
class Rectangle extends Figure {  
 double length;  
 double width;  
 Rectangle(double length, double width) {  
 this.length = length;  
 this.width = width;  
 }  
 @Override double area() { return length \* width; }  
}

## Instructor Screencast: TITLE

# In class 1 (MLO 1, 2, 3) [.5 hours]

Consider Bloch's InstrumentedHashSet, InstrumentedSet, and ForwardingSet examples:

public class InstrumentedHashSet<E> extends HashSet<E>{  
 private int addCount = 0;  
 public InstrumentedHashSet() {}  
  
 @Override public boolean add(E e){  
 addCount++;  
 return super.add(e);  
 }  
 @Override public boolean addAll(Collection<? extends E> c){  
 // What to do with addCount?  
 return super.addAll(c);  
 }  
 public int getAddCount(){ return addCount; }  
}  
  
public class InstrumentedSet<E> extends ForwardingSet<E>{  
 private int addCount = 0;  
  
 public InstrumentedSet(Set<E> s){ super(s); }  
 @Override public boolean add(E e){ addCount++; return super.add(e); }  
 public int getAddCount(){ return addCount; }  
}  
  
public class ForwardingSet<E> implements Set<E> {  
 private final Set<E> s;  
  
 public ForwardingSet(Set<E> s){ this.s = s; }  
  
 public boolean add(E e) { return s.add(e); }  
 public boolean remove(Object o){ return s.remove(o); }  
 @Override public boolean equals(Object o){ return s.equals(o); }  
 @Override public int hashCode() { return s.hashCode(); }  
 @Override public String toString() { return s.toString(); }  
 // Other forwarded methods from Set interface omitted  
}

Consider also the following client code:

Set<String> r = new HashSet<String>();  
r.add("ant"); r.add("bee");  
//r=[a,b]   
  
Set<String> sh = new InstrumentedHashSet<String>();  
sh.addAll(r);  
//sh = [a,b] , sh.getAddCount() = 2  
  
Set<String> s = new InstrumentedSet<String>(r);  
//s = [a,b], s.getAddCount() = 0  
//r = [a,b]  
  
s.add("ant"); s.add("cat");  
//s = [a,b,c] //s.getAddCount() = 2  
//r = [a,b,c]  
//sh = [a,b]  
  
Set<String> t = new InstrumentedSet<String>(s);  
//t = [a,b,c] // t.getAddCount() = 0  
//r = [a,b,c]  
//s = [a,b,c]  
//sh = [a,b]  
  
t.add("dog");  
//t = [a,b,c,d] //t.getAddCount() = 1  
//r = [a,b,c,d]  
//s = [a,b,c,d]  
  
r.remove("bee");  
//r = [a,c,d] ; s = [a,c,d] ; t = [a,c,d]   
  
s.remove("ant");  
//r=s=t=[c,d]

1. How do you think the addCount variable should be updated in the addAll() method in InstrumentedHashSet?
   1. Why is this a hard question?
   2. What does the answer say about inheritance?
   3. Does equals() behave correctly in InstrumentedHashSet?
2. Given your previous answer, what is the value of sh.addCount at the end of the computation?
3. Consider the InstrumentedSet solution. Besides being correct (always a plus!) why is it more general than the InstrumentedHashSet solution?
4. At the end of the computation, what are the values of: r, s, and t?
5. What would a call to s.getAddCount() return at the end of the computation?
6. At the end of the computation, what are the values of: r.equals(s), s.equals(t), and t.equals(s)?
   * Are there any problems with the equals() contract?
7. Would this still work if you globally replaced sets with lists?
8. Would this still work if you globally replaced sets with collections?

**Note**: There is a lot going on in this example. I highly recommend that you play with the code until you understand it.

# Exercise 2 (MLO 1, 2, 3) [.5 hours]

public class Super {  
 public Super() {  
 overrideMe();  
 }  
  
 public void overrideMe () {  
 }  
}  
public final class Sub extends Super {  
  
 private final Date date; // filled in by constructor  
  
 public Sub() {  
 date = new Date();  
 }  
 @Override public void overrideMe () {  
 System.out.println(date);  
 }  
  
 public static void main (String[] args) {  
 Sub sub = new Sub();  
 sub.overrideMe();  
 }  
}

1. What is the pattern, and how common is it?
2. What does the main method do, and why?
3. Which of Bloch's rules does this example break?
4. What does this example mean for Cloneable interface and the clone() method?
5. What does this example mean for Serializable interface and the readObject() method?
6. To what extent does this rule generalize to producer methods?

# Exercise 3 (MLO 1, 2, 3) [.5 hours]

Consider a mutable complex number class:

public class MComplex {  
 double re; protected double im;  
  
 public MComplex (double re, double im) { this.re = re; this.im = im; }  
  
 public double getReal() { return re; }  
 public double getImaginary() { return im; }  
  
 public void setReal(double re) { this.re = re; }  
 public void setImaginary(double im) { this.im = im; }  
  
 public void add (MComplex c) { re += c.re; im += c.im; }  
  
 public void subtract (MComplex c) { re -= c.re; im -= c.im; }  
  
 public void multiply (MComplex c) {  
 double r = re \* c.re - im \* c.im;  
 double i = re \* c.im + im \* c.re;  
 re = r; im = i;  
 }  
  
 public void divide (MComplex c) {  
 double den = c.re \* c.re + c.im \* c.im;  
 double r = (re \* c.re - im \* c.im) / den;  
 double i = (re \* c.im + im \* c.re) / den;  
 re = r; im = i;  
 }  
  
 @Override public boolean equals (Object o) {  
 if (o == this) return true;  
 if (!(o instanceof MComplex)) return false;  
 MComplex c = (MComplex) o;  
  
 // See Bloch page 43 to find out why to use compare() instead of ==  
 return Double.compare(re, c.re) == 0 &&  
 Double.compare(im, c.im) == 0;  
 }  
  
 @Override public int hashCode () {  
 int result = 17 + hashDouble(re);  
 result = 31 \* result + hashDouble(im);  
 return result;  
 }  
  
 private int hashDouble (double val) {  
 long longBits = Double.doubleToLongBits(val);  
 return (int) (longBits ^ (longBits >>>32));  
 }  
  
 @Override public String toString() { return "(" + re + " + " + im + "i)"; }  
}

Before we get to immutability, consider the method contracts. Where do the various contracts "come from", and is there anything in the (missing) JavaDoc that might require a bit of research?

Apply each of Bloch's 5 rules for making a class immutable:

1. Don't provide any methods that modify the object's state. How do you handle the mutators?
2. Ensure that no methods can be overridden.
   * Why is this a problem? Show me!
   * Fix the problem:
     + Change the class declaration, or
     + Change the method declarations, or
     + Change the constructor visibility.
3. Make all fields final.
4. Make all fields private.
   * Is there a significant difference in visibility between re and im?
5. Ensure exclusive access to any mutable components.

# Module 3 Assignment – (MLO 1, 2, 3) [~1 hours]

## Purpose

We will practice the advice from Item 18 in Bloch "Favoring composition over inheritance"

## Instructions

Consider the InstrumentedSet example from Bloch Item 18 (i.e., in-class exercise 1).

1. Replace Set with List. There is no problem with equals(). Why not?
2. Replace Set with Collection. Now equals() does not satisfy its contract.
   * Explain why there is a problem.
   * Demonstrate the problem with a suitable JUnit test.

### Grading Criteria

The GTA will look for correct responses, appropriate JUnit tests, and plausible explanations when doing the grading.

## Deliverable

* Submit a .java file for your implementation.

## Due Date

Your assignment is due by Sunday 11:59 PM, ET.

# Module 1 Quiz (MLO 1, 2, 3) [~.5 hour]

## Purpose

Quizzes in this course give you an opportunity to demonstrate your knowledge of the subject material.

## Instructions

Consider the following code, and suppose the main method in Sub is executed.

public class Super {  
 private String y;  
 public Super () { stut();}  
 public void stut() { if (y == null) {y = "cat";} else {y = y + y;}}  
}  
public class Sub extends Super {  
 private String x;  
 public Sub (String s) { x = s;}  
 @Override public void stut() {  
 x = x + x;  
 }  
 public static void main(String[] args) {  
 Super s = new Sub("dog");  
 }  
}

1. Is the constructor in Super invoked? Why or why not? **Answer**: Yes. There is an implicit call to super(); as the very first instruction in the Sub constructor
2. Is the stut() method in Super invoked? Why or why not? **Answer**: No. The stut() method is overridden in class Sub, and so the class Sub stut() method is invoked instead.
3. Is the stut() method in Sub invoked? Why or why not? **Answer**: Yes. See the answer to the previous question.
4. Which rule in Bloch does this code violate? (Any reasonable phrasing of the principle is fine.) **Answer**: The rule that says "Constructors (in classes that may be extended) must not invoke overridable methods."

The quiz is 30 minutes in length. The quiz is closed-book.

## Deliverable

Use the link above to take the quiz.

## Due Date

Your quiz submission is due by Sunday 11:59 PM, ET.