Object Oriented Design and UML

- Software Development Activities
- Object Oriented Design
- Unified Modeling Language (UML)
- Reading for this Lecture: L&L 6.1 – 6.3

Software Development

- Software involves four basic activities:
  1. Establishing the requirements
  2. Creating a design
  3. Implementing the code
  4. Testing the implementation
- These activities are not strictly linear – they overlap and interact
- We’ve already done a lot with #3 and #4
- Now, we’ll concentrate on #1 and #2

Requirements

- *Software requirements* specify the tasks that a program must accomplish
  - what to do, not how to do it
- Often an initial set of requirements is provided, but they should be critiqued and expanded
- It is difficult to establish and document detailed, unambiguous, and complete requirements
- Careful attention to the requirements can save significant time and expense in the overall project

Design

- A *software design* specifies how a program will accomplish its requirements
- That is, a software design determines:
  - how the solution can be broken down into manageable pieces
  - what each piece will do
- An object-oriented design determines which classes and objects are needed and specifies how they will interact
- Low level design details include how individual methods will accomplish their tasks

Object-Oriented Design

- Design Methodology / Process
  - Analyze / decompose the requirements
  - Determine the classes required for a program
  - Define the relationships among classes
- Tool: Unified Modeling Language (UML)
  - Use Case Diagram
  - Class Diagram
  - Interaction Diagram

Identifying Classes and Objects

- The core activity of object-oriented design is determining the actors, classes, and objects that represent the problem and its solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

Identifying Classes and Objects

- A partial requirements document:

The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Of course, not all nouns will correspond to an actor, class or object in the final solution

Identifying Classes and Objects

- A class represents a group (a "classification") of objects with the same attributes and behaviors
- Generally, classes that represent objects should be given names that are singular nouns
- Examples: `Coin`, `Student`, `Message`
- A class represents the concept of one such object
- We are free to instantiate as many "instances" of each object as needed
- Good selection of object names for the instances can be helpful to understanding

Identifying Classes and Objects

- Sometimes it is challenging to decide whether something should be represented as a class
- For example, should an employee's address be represented as a set of instance variables or as an `Address` object
- The more you examine the problem and its details the more clear these issues become
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

Identifying Classes and Objects

- We want to define classes with the proper amount of detail
- For example, it may be unnecessary to create separate classes for each type of appliance in a house
- It may be sufficient to define a more general `Appliance` class with appropriate instance data
- It all depends on the details of the problem being solved

Identifying Classes and Objects

- Part of identifying the classes we need is the process of *assigning responsibilities* to each class
- Every activity that a program must accomplish must be represented by one or more methods in one or more classes
- We generally use verbs for the names of methods
- In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design

Unified Modeling Language (UML)

- UML is a graphical tool to visualize and analyze the requirements and do design of an object-oriented solution to a problem
- Three basic types of diagrams:
  - Use Case Diagram
  - Class Diagram
  - Interaction Diagram
- A good reference is *UML Distilled, 3rd Ed.*, Martin Fowler, Addison-Wesley/Pearson

Unified Modeling Language (UML)

- Advantage of UML – It is graphical
  - Allows you to visualize the problem / solution
  - Organizes your detailed information
- Disadvantage of UML – It is graphical
  - Can be done with pencil and paper - tedious
  - We have UMLPAD which is a simple design tool to aid in drawing the diagrams
  - Commercial UML S/W tools may be expensive!
    - Example: Rational ROSE (IBM acquired Rational)

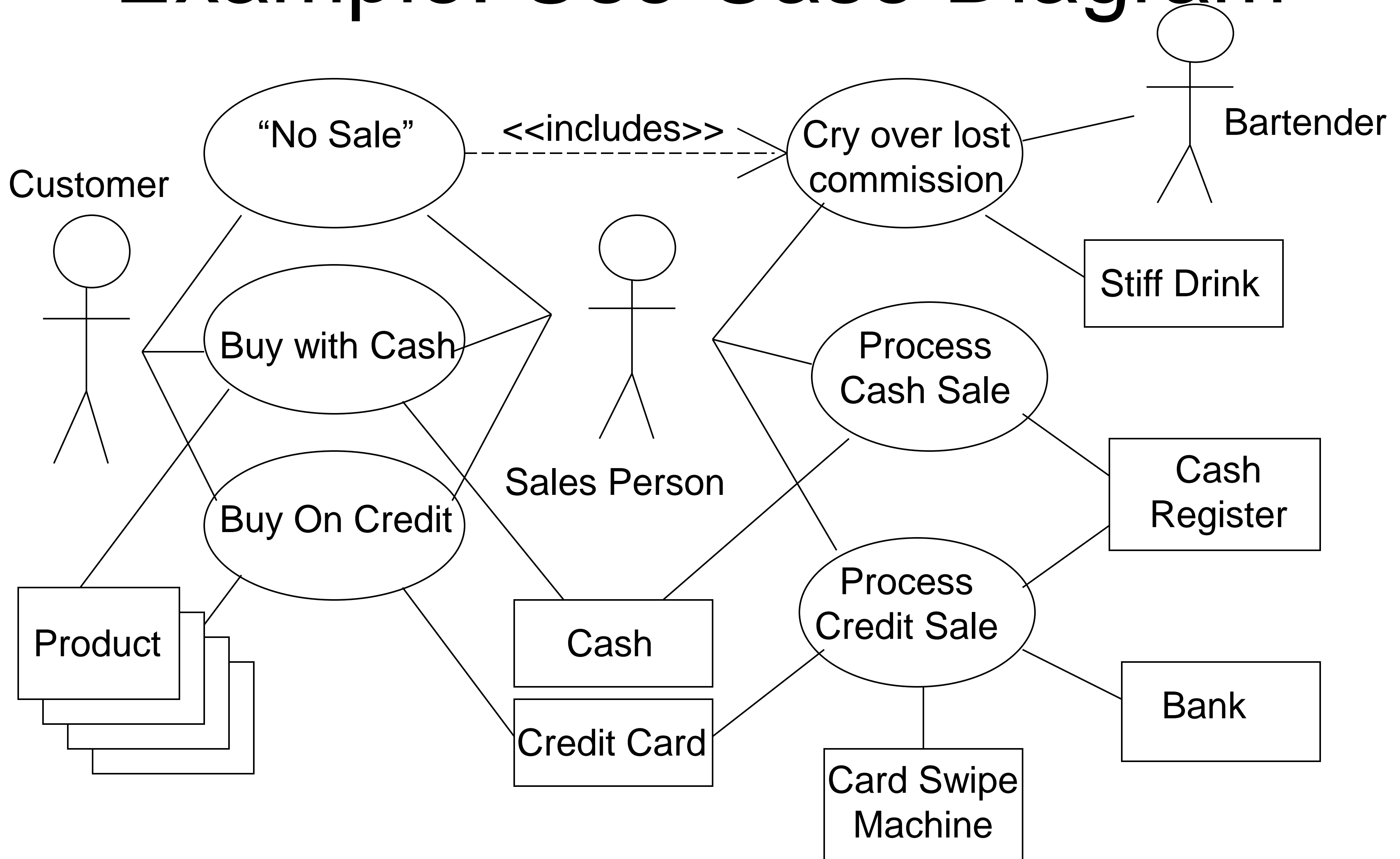
Use Case Diagrams

- Typically the first diagram(s) drawn
- Helpful for visualizing the requirements
- Icons on the Use Case Diagram
  - Actors: Users or other external systems
  - Objects: Potential classes in the solution
  - Scenarios: Sequences of interactions between Actors and Objects that are typical for the solution to the problem (Both success cases and error cases should be included)

Example: Use Case Diagram

- Actors: Sales person, Customer, Bartender
- Objects: Products, Cash, Cash Register, Credit Card, Card Swipe Machine, Bank
- Scenarios involving Actors and Objects:
  - Customer listens to sales pitch but doesn't buy
  - Customer buys product with cash
  - Customer buys product with credit card
    - Success scenario: Bank accepts the card
    - Error scenario: Bank says card is "maxed out"

Example: Use Case Diagram



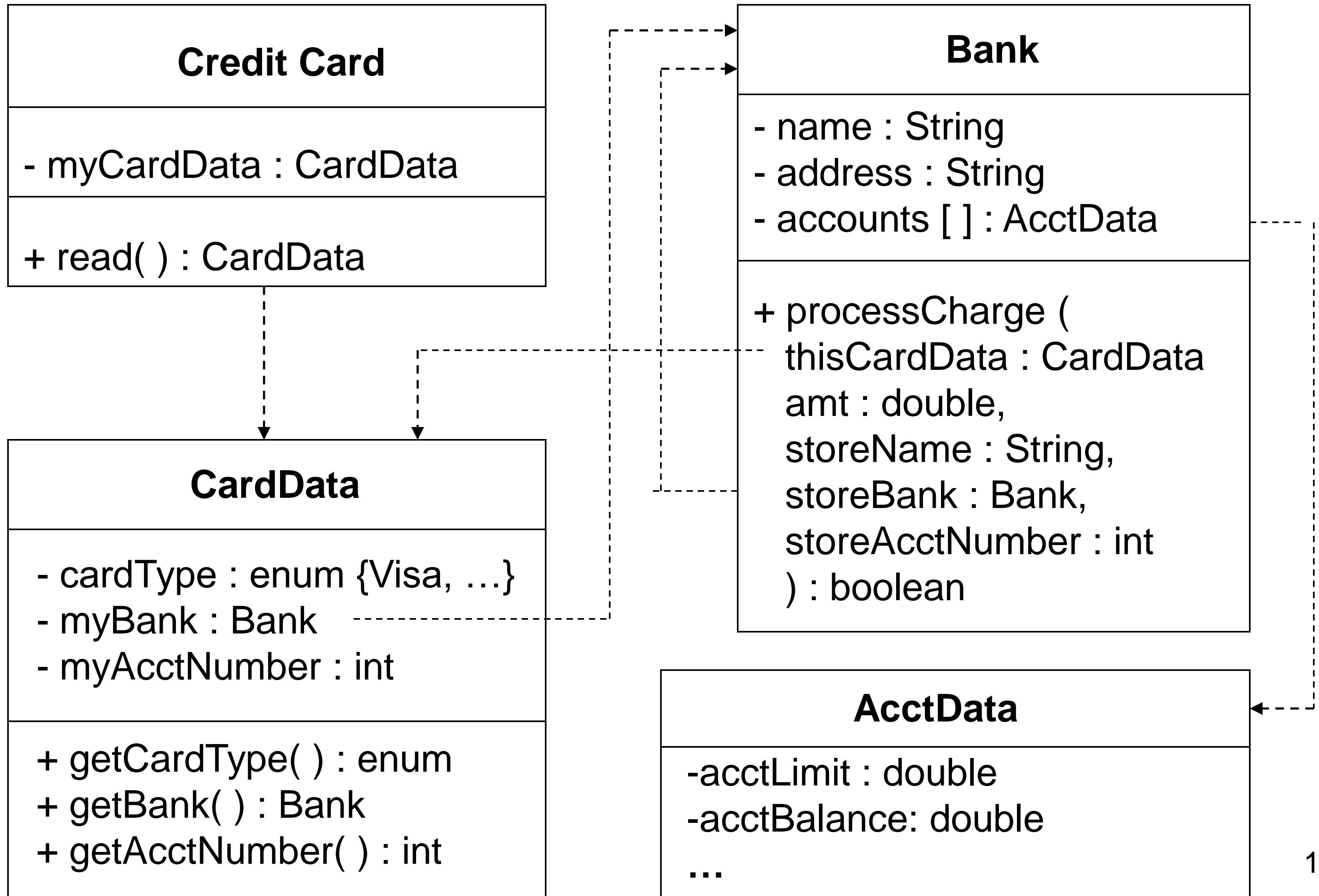
Example: Scenario

- Process Credit Sale
  - Swipe Card
  - Enter Amount of Sale
  - Wait for Bank Response
  - Success Variation (Bank accepts charge)
    - Record authorization number
    - Get customer signature
    - Give customer product(s) and receipt
  - Error Variation (Card maxed out)
    - Inform Customer that card was rejected
    - <<include>> Cry over Lost Commission

Class Diagrams

- Classify the Objects in the Use Cases
- Define name of each class
- Define each class's attributes
  - Constants
  - Variables
- Define each class's behaviors
  - Methods
- Show relationships between classes
  - Depends on, Inherits, etc.

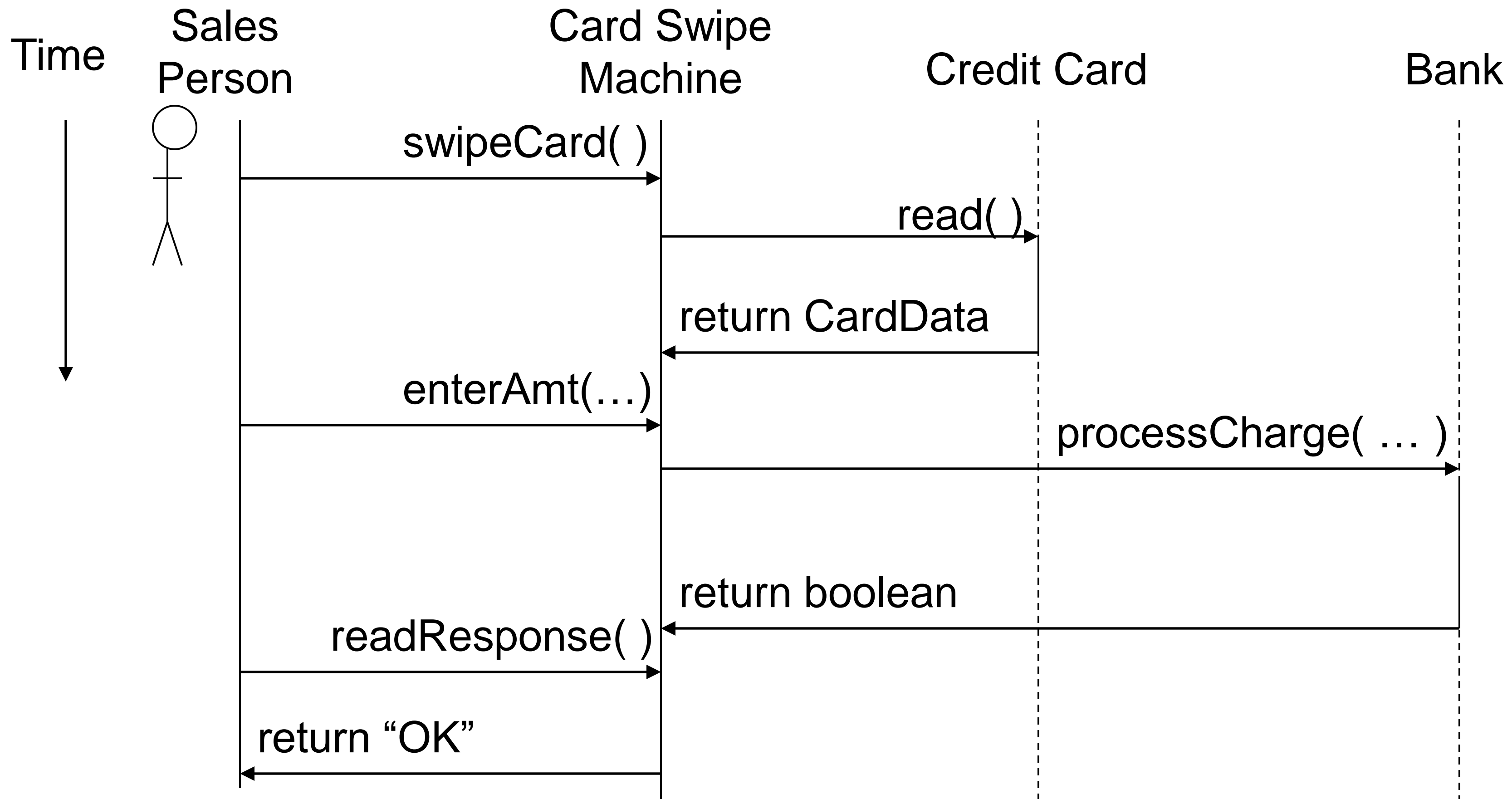
Example: Class Diagram



Interaction Diagrams

- Shows the time relationship of the events in a scenario between actors and objects
  - UML Sequence Diagram
  - Sometimes called a “ladder diagram”
- A vertical line represents an actor or object
- A horizontal line represents an interaction
  - E.G. a call to a method of another object
- Progress of time is shown down the page

Example: Interaction Diagram

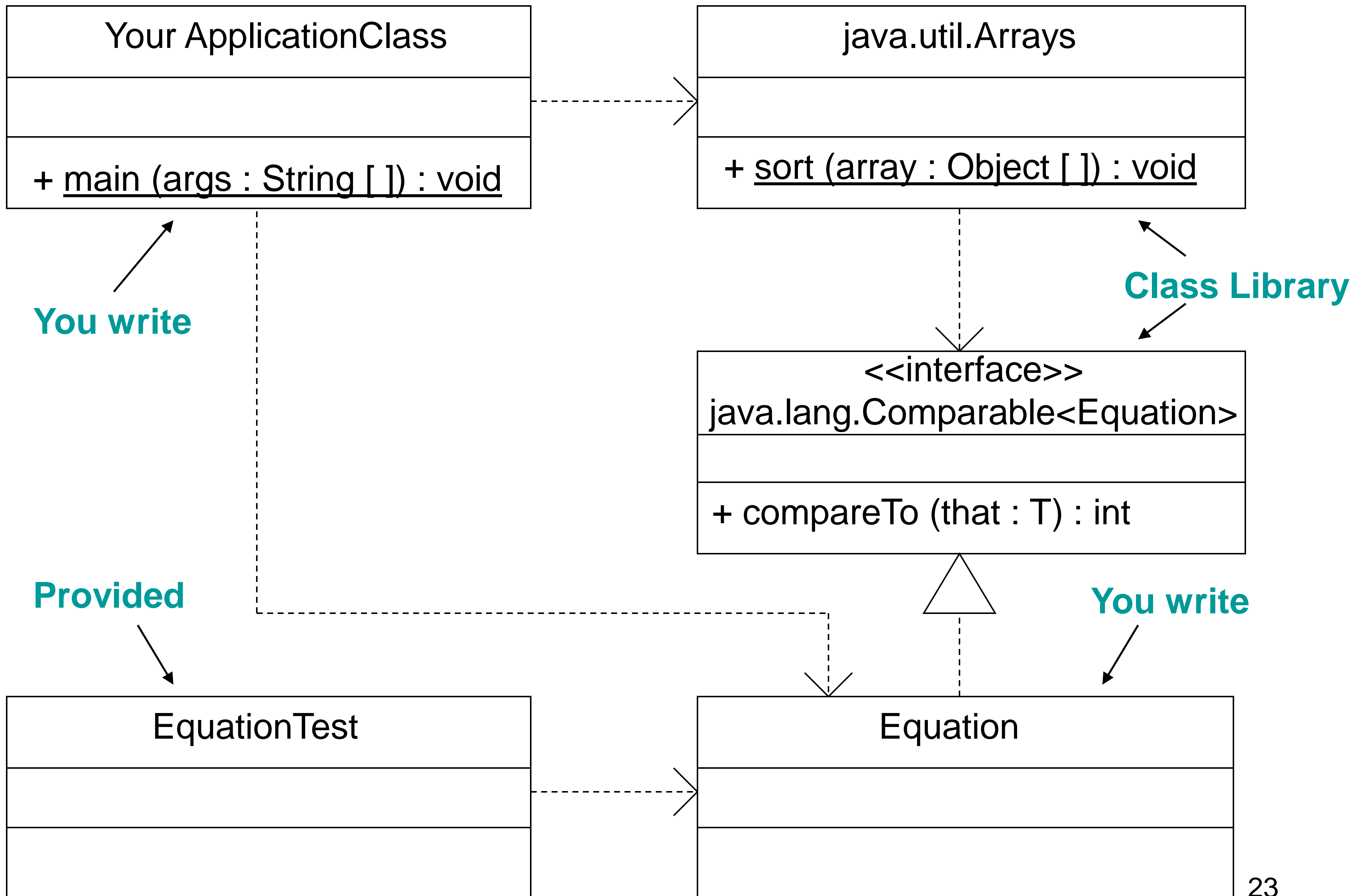


Introduction to Project 3

- In Project 3, you will write a class that encapsulates the three integer coefficients of a linear equation:

$$ax + by = c$$

- The methods are defined in the assignment
- The class implements the interface Comparable<T> (we'll cover implementing interfaces shortly)



Your ApplicationClass
+ main (args : String [ ]) : void
java.util.Arrays
+ sort (array : Object [ ]) : void
You write
Class Library
<<interface>>
java.lang.Comparable<Equation>
+ compareTo (that : T) : int
Provided
You write
EquationTest
Equation

Equation Class UML Diagram

Equation implements Comparable<Equation>
+ THRESHHOLD : double // threshold for comparisons of double values - a : int // coefficient of the variable x - b : int // coefficient of the variable y - c : int // constant

+ Equation(a : int, b : int, c : int) // constructor
+ toString( ) : String // return a String representing the equation
+ slope( ) : double // return the slope of the line
+ intercept( ) : double // return the y-axis intercept of the line
+ compareTo(that : Equation) : int // compare values of equations
// solving methods for pairs of equations:
+ solveForXWith(that : Equation) : double // solve for X
+ solveForYWith(that : Equation) : double // solve for Y
+ verifySolution(x : double, y : double) : boolean // verify correct