UML Class Modeling

* UML is the Unified Modeling Language
* It is about communication. Specifically, it allows us to communicate structure, behaviour, and interaction.

**WHAT IS CLASS MODELING**

The words “class” and “object” are often used as if they are the same, but actually they are not. An object is someone or something, like the person “John” or this document. It can be concrete like “The Eiffel Tower” or abstract like “France”. Formally put, an object is something with properties, relationships, behavior and an identity.

A class is a definition of objects that share the same kind of properties, relationships and behavior, like the class Employee may define the properties “name”, “age” and “employee number” and may define a relationship with the class Department. An object is a specific instance of a class, like “John” may be an instance of the class Employee.

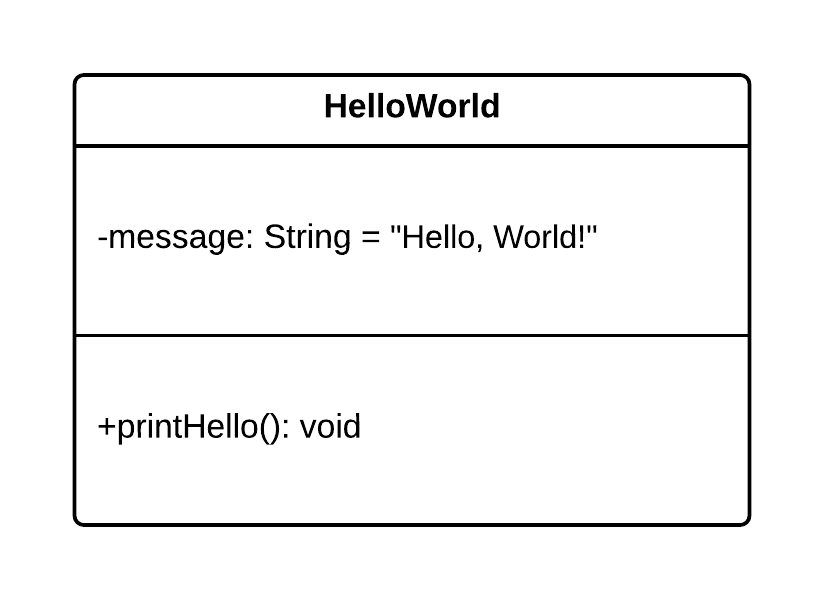
**Class Modeling** is the task of specifying these classes using a specific language in which the *properties* are called *attributes*, the *relations* are called *associations* and *behavior* is defined as *operations*.

**Class Diagrams - Basic Concepts**

Class diagrams represent object-oriented classes, with fields, methods, relationships, etc.

1. Used to model the type of objects and relationships among them.
2. One of the most commonly used UML diagram.
3. Drawn by developers as a part of design activity.

UML Class Diagrams consists of three sections: the **class name**, **attributes**, and **operations**.



* Class names are written in **bold.**
* Abstract class names are written in *italics.*
* Static attributes and operations are underlined.

**Attributes** refer to the properties, or fields, or a class.  They are expressed as follows:

<visibility><name>: <type> = <default value> <{modifier}>

* **Visibility**refers to the access modifiers on the attributes, and UML supports the following access modifiers (relate them back to the programming language in which you will write your code):
  + **+**(public)
  + **-** (private)
  + **~** (package)
  + **#** (protected)
* The **name**of an attribute is used to identify the attribute.
* The **type** of the attribute refers to the data type that it will use (String, Integer, etc.).
* The **default value** of an attribute is optional, and may be left blank.
* The **modifier**is optional, and serves to provide additional information on the attribute, such as {readOnly}.

**Operations** refer to the behaviours, or methods of a class.  They are expressed as follows:

<visibility><name>(<parameter list>): <return type>

* **Visiblity**and **name** are self-explanatory - they serve the same purpose as they do on attributes.
* The **parameters**refer to the parameters (or method arguments) that will be passed to the operation when it is invoked.  They are represented as follows: <parameter name>: <parameter type>
* The **return type** refers to the type of the result returned by the operation, such as String or Integer (or void, if it does not return anything).

Expressed in Java code, the class above looks like this.

1. public class HelloWorld {
3. private String message = "Hello, World!";
5. public void printHello() {
6. System.out.println(message);
7. }
8. }

**Class Diagrams - Relationships**

Object-oriented classes have relationships between them - they extend each other (**inheritance**), they depend on each other (**dependency**), they interact with each other (**association**), or they form part of each other (**aggregation**and **composition**).

Relationships also have **multiplicity**, which shows how many instances of a class can exist on each side of a relationship.

**Dependency** relationships exist when classes depend on each other in such a way that a change to one class may affect the other, such as when one class accepts an instance of another class as parameter to a method.

* Dependency relationships are represented by arrows on dashed lines.  Stereotypes can be indicated within guillemets to provide further detail on the nature of the relationship.

A black line on a white background

Description automatically generated  
A black text with a white background

Description automatically generated

* A dependency relationship can exist when we have a Library class that manages Book objects.  Since the Library class has a method that returns a Book, changes to the Book class could result in changes to the Library class (based on how Book objects are created).

A diagram of a diagram

Description automatically generated

In code, this looks as follows.

1. public class Library {
3. public Book findBook(String name) {
4. //Do some book stuff.
5. return new Book();
6. }
7. }
9. class Book {
11. private String name;
12. private String author;
13. private String isbn;
15. //Getters/setters omitted.
16. }

**Generalization**relationships exists when one class extends another class (making it a specialization of the parent class, like a Car is a specialization of a Vehicle).

* Generalization relationships are represented by a triangular arrow on a solid line.  
  A black arrow pointing up

  Description automatically generated
* Operations and attributes on the parent class also exist on the child classes, without being explicitly specified.
* A generalization relationship can exist when we have a system that keeps track of vehicle rentals, where we have various specialized vehicles.

A diagram of a vehicle

Description automatically generated

Expressed in code, these classes look like this.

1. public class Vehicle {
3. private String manufacturer;
4. private int yearOfManufacture;
6. public String getManufacturer() {
7. return manufacturer;
8. }
10. public void setManufacturer(String manufacturer) {
11. this.manufacturer = manufacturer;
12. }
14. public int getYearOfManufacture() {
15. return yearOfManufacture;
16. }
18. public void setYearOfManufacture(int yearOfManufacture) {
19. this.yearOfManufacture = yearOfManufacture;
20. }
22. public boolean isAvailableForRental() {
23. //Execute some logic.
24. return true;
25. }
26. }
28. class Truck extends Vehicle {
30. private double engineCapacity;
31. private double loadBinSize;
33. public double getEngineCapacity() {
34. return engineCapacity;
35. }
37. public void setEngineCapacity(double engineCapacity) {
38. this.engineCapacity = engineCapacity;
39. }
41. public double getLoadBinSize() {
42. return loadBinSize;
43. }
45. public void setLoadBinSize(double loadBinSize) {
46. this.loadBinSize = loadBinSize;
47. }
48. }
50. class Boat extends Vehicle {
52. private double length;
53. private int numberOfEngines;
55. public double getLength() {
56. return length;
57. }
59. public void setLength(double length) {
60. this.length = length;
61. }
63. public int getNumberOfEngines() {
64. return numberOfEngines;
65. }
67. public void setNumberOfEngines(int numberOfEngines) {
68. this.numberOfEngines = numberOfEngines;
69. }
71. public static void main(String[] args) {
72. Boat boat = new Boat();
73. boat.setManufacturer("Boat Company");
74. boat.setYearOfManufacture(2018);
75. boat.setLength(5.5);
76. boat.setNumberOfEngines(2);
78. Vehicle vehicle = boat;
79. }
80. }

**Association** relationships often exist when classes have variables of other types, that they can invoke operations on.

* Association relationships are represented by an arrow on a solid line.

A black and white object

Description automatically generated

* Association relationships can be bi-directional, in which case both classes can reference each other.

A black line on a white background

Description automatically generated

* Association relationships can also include multiplicity, where we can have one instance on one side and exactly zero or one instance on the other side, or one instance on one side zero or more instances on the other side (**\*** refers to any number of instances), or any other combination.

A black line on a white background

Description automatically generated  
  
A black line on a white background

Description automatically generated

* An association relationship can exist when we model the relationship between doctors and patients, where a doctor can have any number of patients and a patient can only be treated by one doctor at a time.

A diagram of a patient

Description automatically generated

If we express this diagram in code, it looks like this.

1. public class Doctor {
3. private String name;
4. private String registrationNumber;
5. private List<Patient> patients = new ArrayList<Patient>();
7. public void addPatient(Patient patient) {
8. this.patients.add(patient);
9. }
11. //Getters/setters omitted.
12. }
14. class Patient {
16. private String patientId;
17. private String name;
18. private int age;
20. //Getters/setters omitted.
21. }

**Aggregation**relationships exist when we aggregate (or bring together) objects of one class in another class.

* Aggregation relationships are represented by an unfilled diamond on the 'owning' side of the relationship.

A black arrow with a white background

Description automatically generated

* Objects on both sides of an aggregation relationship can exist in isolation.
* Aggregation relationships can have multiplicity.
* An association relationship can exist when we model teams and players.  A player can exist without belonging to a team, and a team can exist without any players.

A diagram of a game

Description automatically generated

In code, our example looks like this.

1. public class Team {
3. private String name;
4. private String slogan;
5. private String city;
7. private List<Player> players = new ArrayList<>();
9. //Getters/setters omitted.
11. public void addPlayer(Player player) {
12. this.players.add(player);
13. player.setTeam(this);
14. }
16. public void removePlayer(Player player) {
17. this.players.remove(player);
18. player.setTeam(null);
19. }
20. }
22. class Player {
24. private String name;
25. private int age;
26. private String homeTown;
28. private Team team;
30. //Getters/setters omitted.
32. public void setTeam(Team team) {
33. this.team = team;
34. }
35. }

**Composition** relationships when objects are composed of (or made up of) other objects.

* Composition relationships are represented by a filled diamond on the 'owning' side.

A black arrow with a triangle

Description automatically generated

* Objects in a composition relationship cannot, conceptually, exist in isolation.  This isn't always easy to enforce in code.
* If the parent object in a composition relationship is destroyed, so are the child objects.
* A composition relationship can exist when we model a system for creating web pages - pages cannot exist without a page header and a page body, and each PageHeader and PageBody object **must** belong to a WebPage object.

A diagram of a web page

Description automatically generated

If we express the WebPage model above in code, it could look like this.

1. public class WebPage {
3. private final PageHeader header;
4. private final PageBody pageBody;
6. public WebPage(PageHeader header, PageBody pageBody) {
7. this.header = header;
8. this.pageBody = pageBody;
9. }
10. }
12. class PageHeader {
14. private String title;
15. private String charset;
17. //Getters/setters omitted.
18. }
20. class PageBody {
22. private String body;
24. //Getters/setters omitted.
25. }

**Class Diagrams - Advanced Concepts**

**Notes**are used to add additional information to UML Class Diagrams (and other UML diagrams).

* Notes are represented by a rectangle with the top-right corner folded over.
* Notes can exist on the diagram itself, or they can be linked to specific elements with a dashed line.

A diagram of a computer

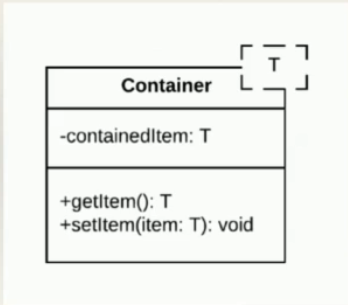
Description automatically generated

**Template classes** are parameterized to operate on specific types.

* Template classes are represented by a generic type in a dashed block on the top-right corner of the class.



* Template classes correspond to generic classes in languages like Java and C#.
* A template class can be used to represent a Container class that can hold objects of a specific type.



1. public class Container<T> {
3. private T containedItem;

6. private T getItem(){
7. return containedItem;
8. }
10. public void setItem<T item> {
11. this.containedItem = item
12. }
13. }

**Derived attributes** are class attributes that can be calculated based on other values on a class (or outside of it, such as the current date).

* Derived attributes are indicated by a forward slash (/) preceding the attribute name (e.g. /age: int ).

A close-up of a card

Description automatically generated

* We can use notes to indicate how derived attributes are calculated (although we could also show this on a sequence- or activity diagram).
* We can use a derived attribute to model a class that calculates a customer's age.

A close-up of a box

Description automatically generated

In code, our derived attribute could be calculated as follows.

1. public class Customer {
3. private final String name;
4. private final LocalDate dateOfBirth;
6. private final int age;
8. public Customer(String name, LocalDate dateOfBirth) {
9. this.name = name;
10. this.dateOfBirth = dateOfBirth;
12. LocalDate now = LocalDate.now();
14. this.age = now.getYear() - dateOfBirth.getYear();
15. }
17. //Getters omitted.
19. }

**Keywords** are used to provide additional metadata on a class or class element.

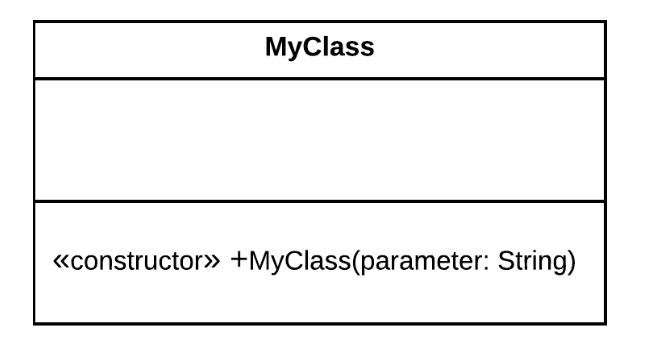
* Keywords on a class (the ones that we use in this course) are indicated in guillemets.

A close-up of a keyword

Description automatically generated

**Constructors** are special operations used to create instances of a class.

* Constructors share the name of the class and do not specify a return type.
* Constructors can be indicated with the 'constructor' keyword.



* Constructors can also be indicated without the 'constructor' keyword, since one can easily infer that an operation is a constructor.

A white rectangular object with black text

Description automatically generated

*EXTRA MATERIALS****:***



**Classifier**

An abstract metaclass whose concrete subclasses classify different types of values

* Concrete Class
* Abstract Class
* Interface
* Enumerations
* Generic Class

**Feature**

Structural and behavioral characteristics of a classifier

* Structural

Properties or attributes

* Behavioral

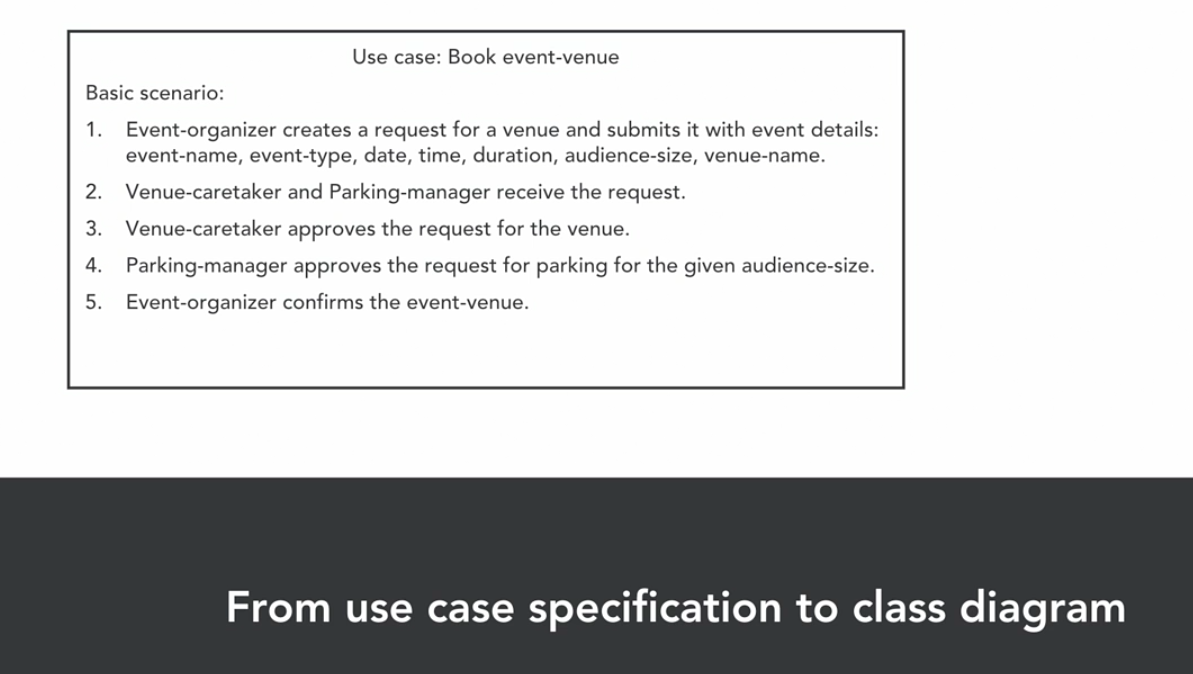
Operations or methods

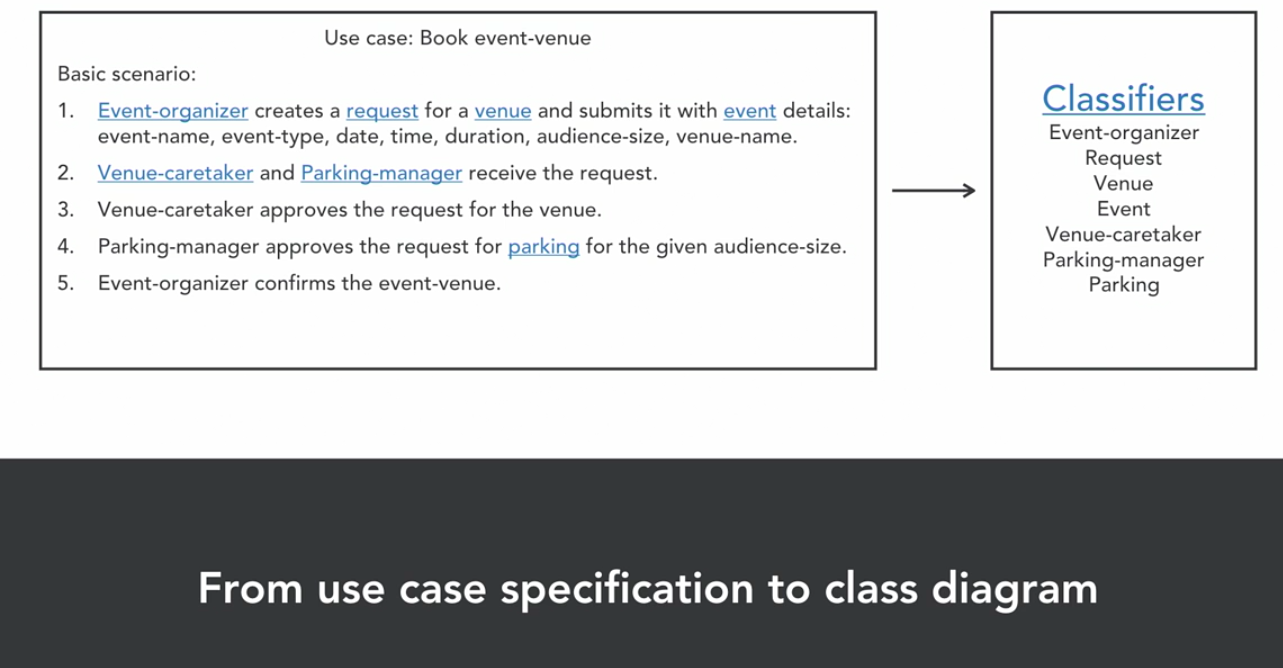
**Relationship**

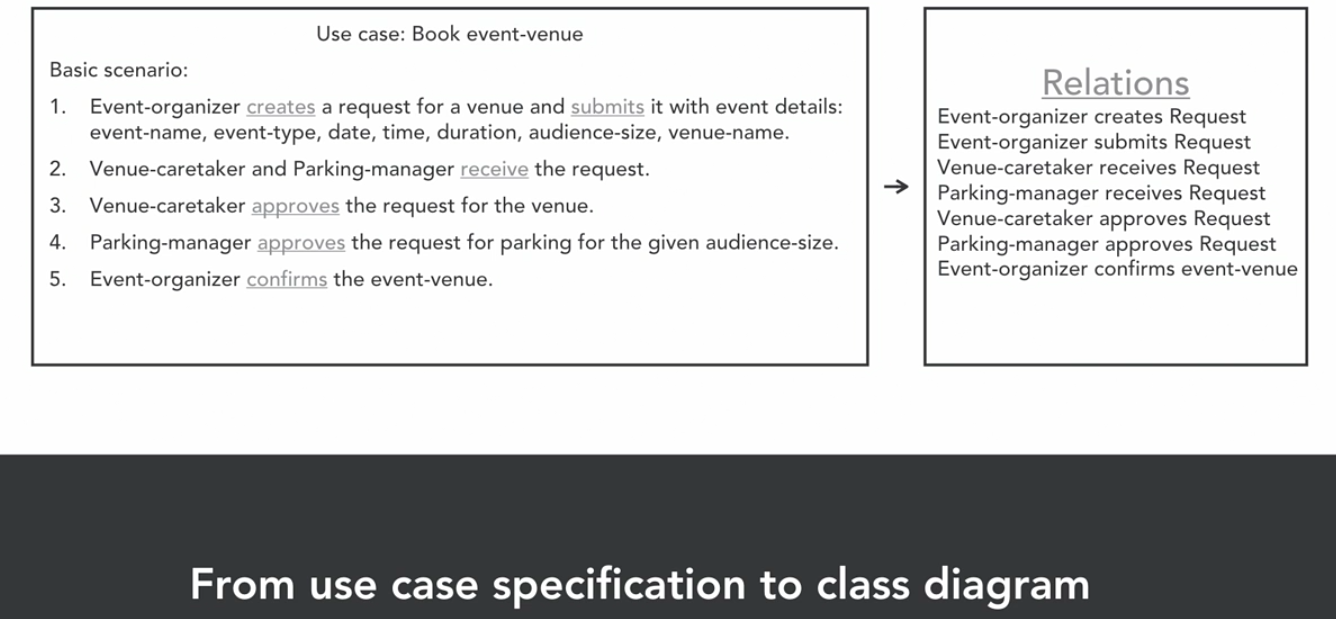
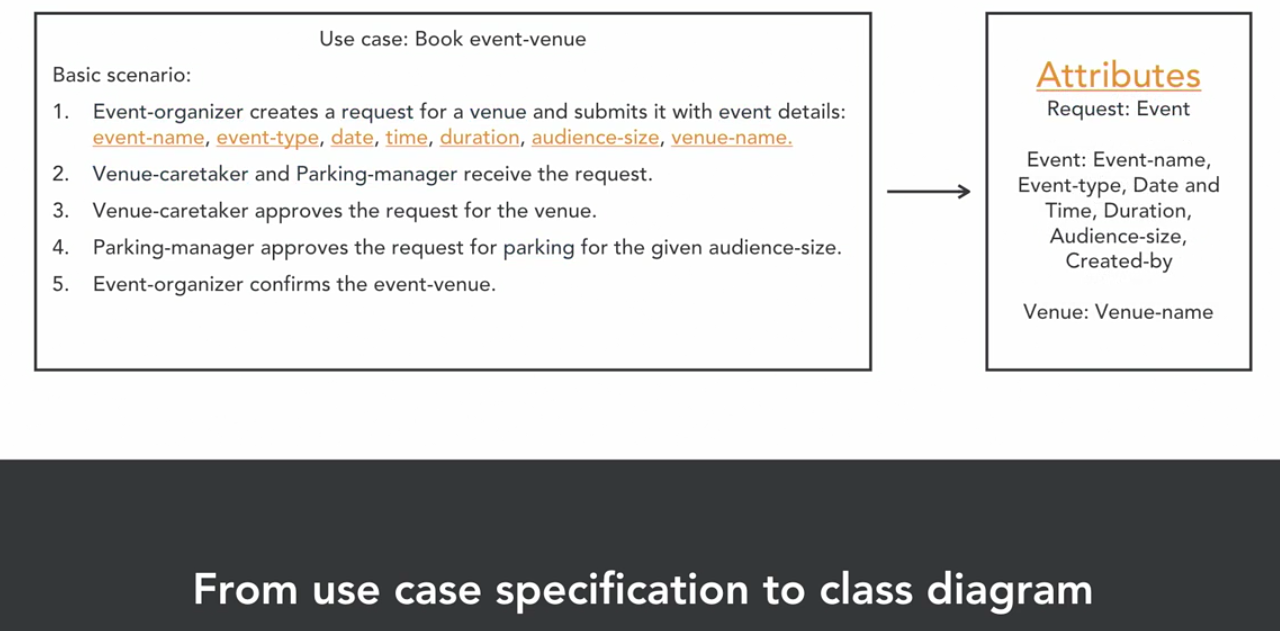
Relationship among classifiers

* Association
* Generalization (Inheritance)
* Dependency ---->

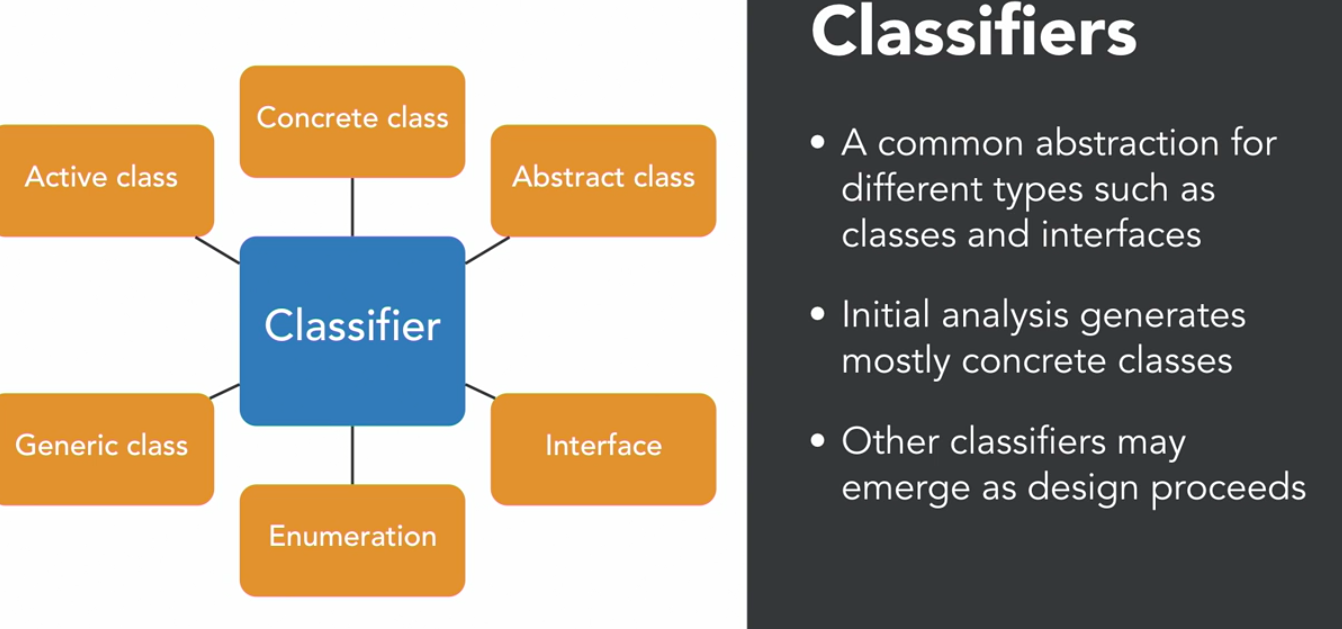
A sample scenario:

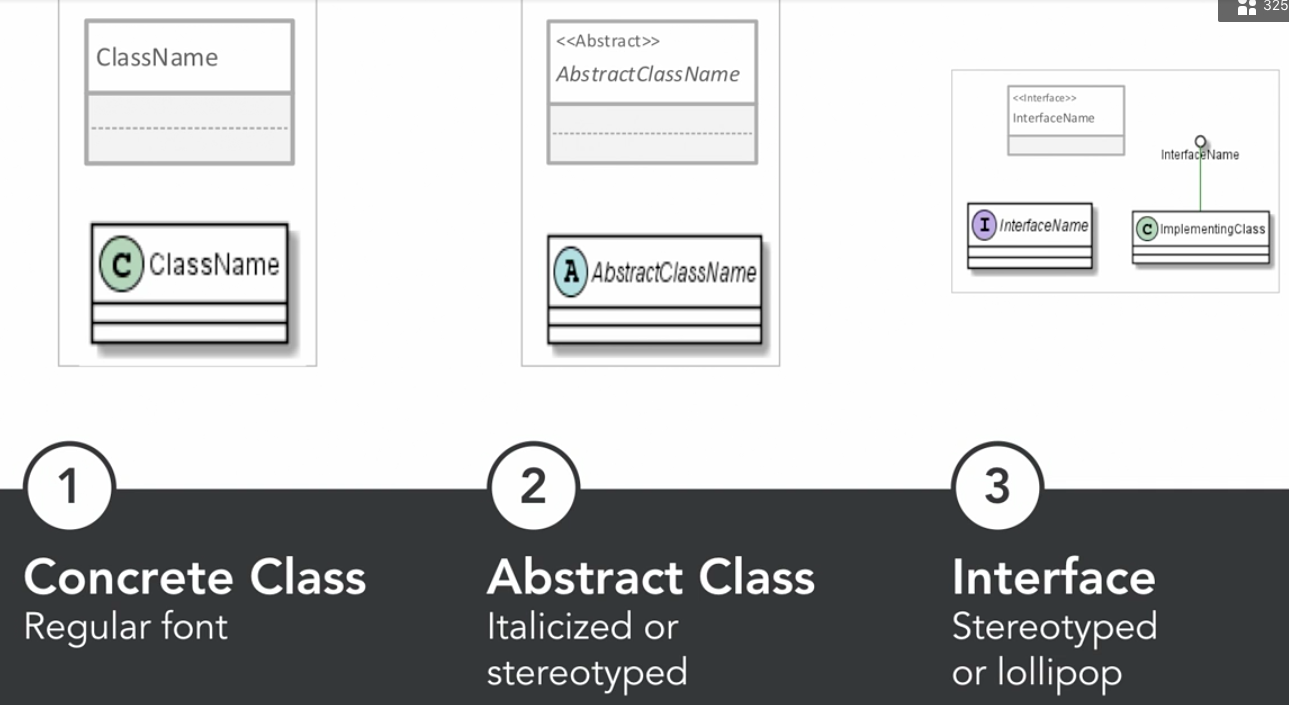


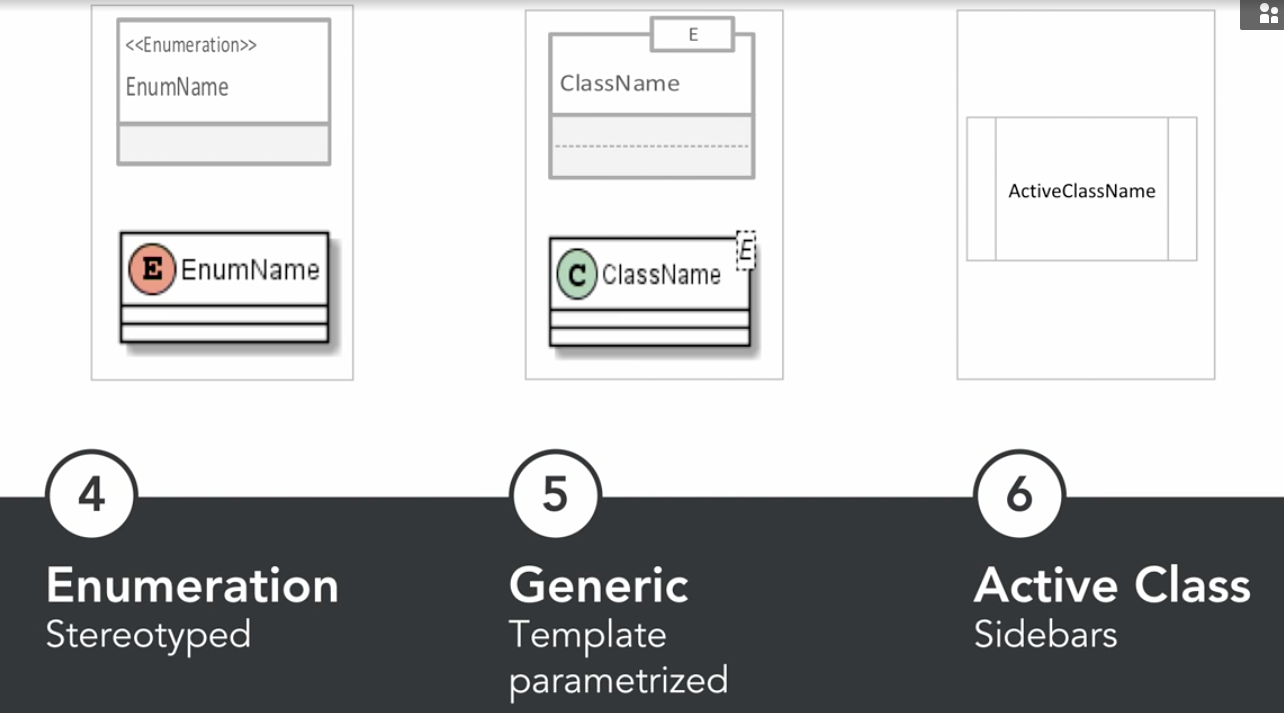




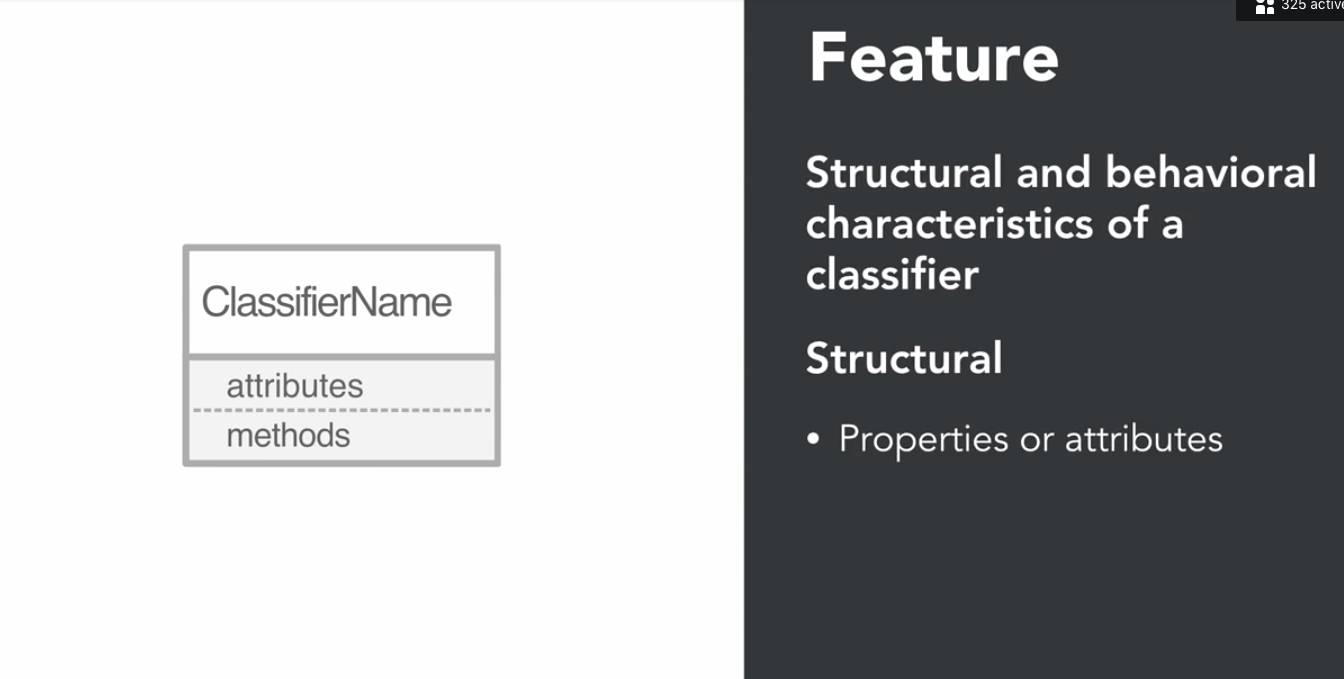
CLASSIFIERS

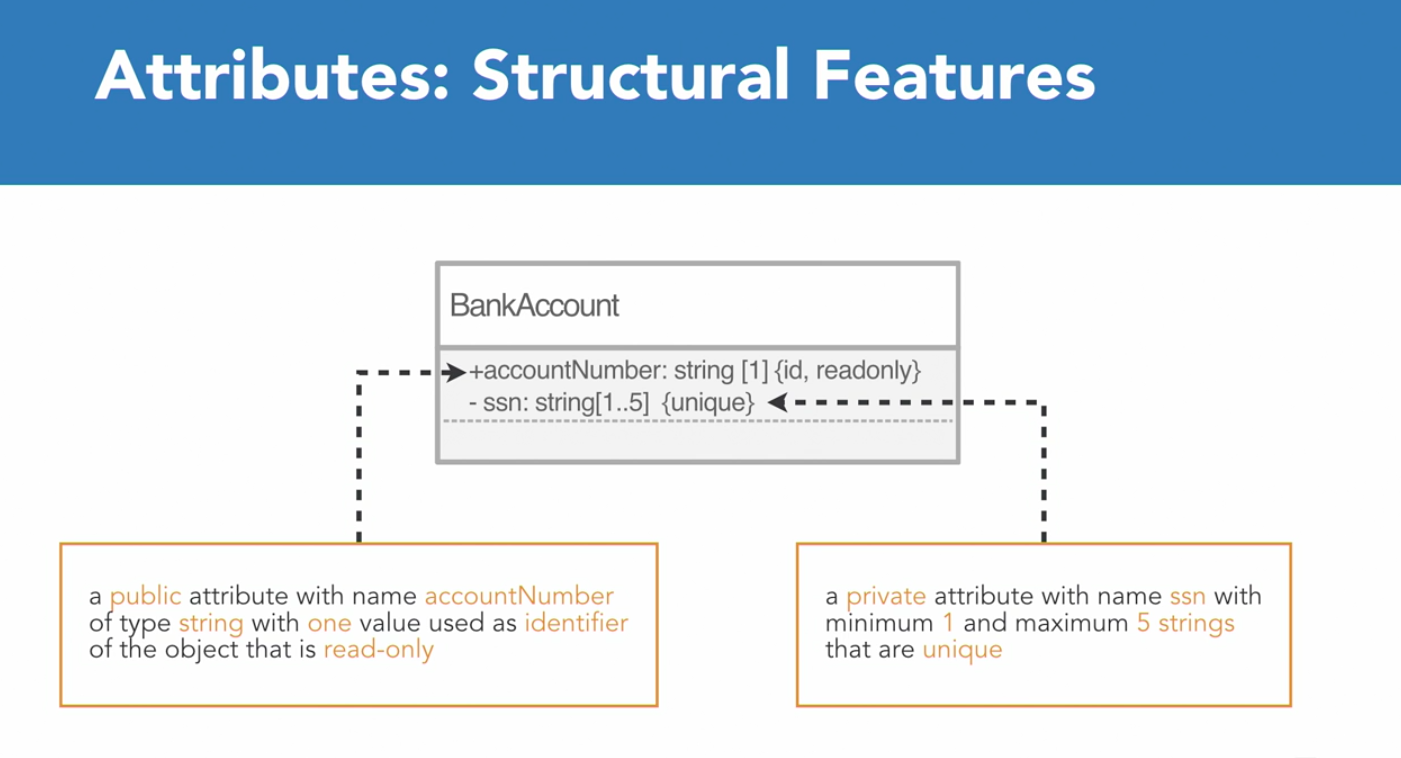


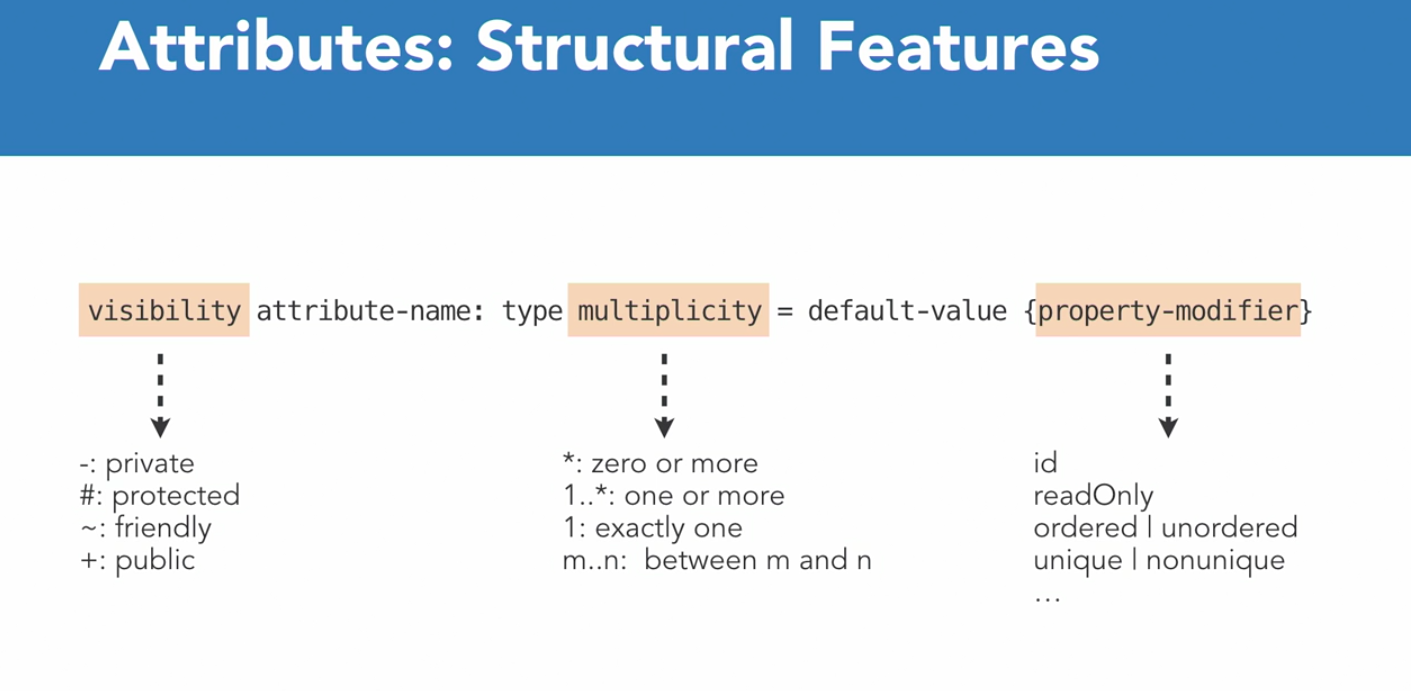




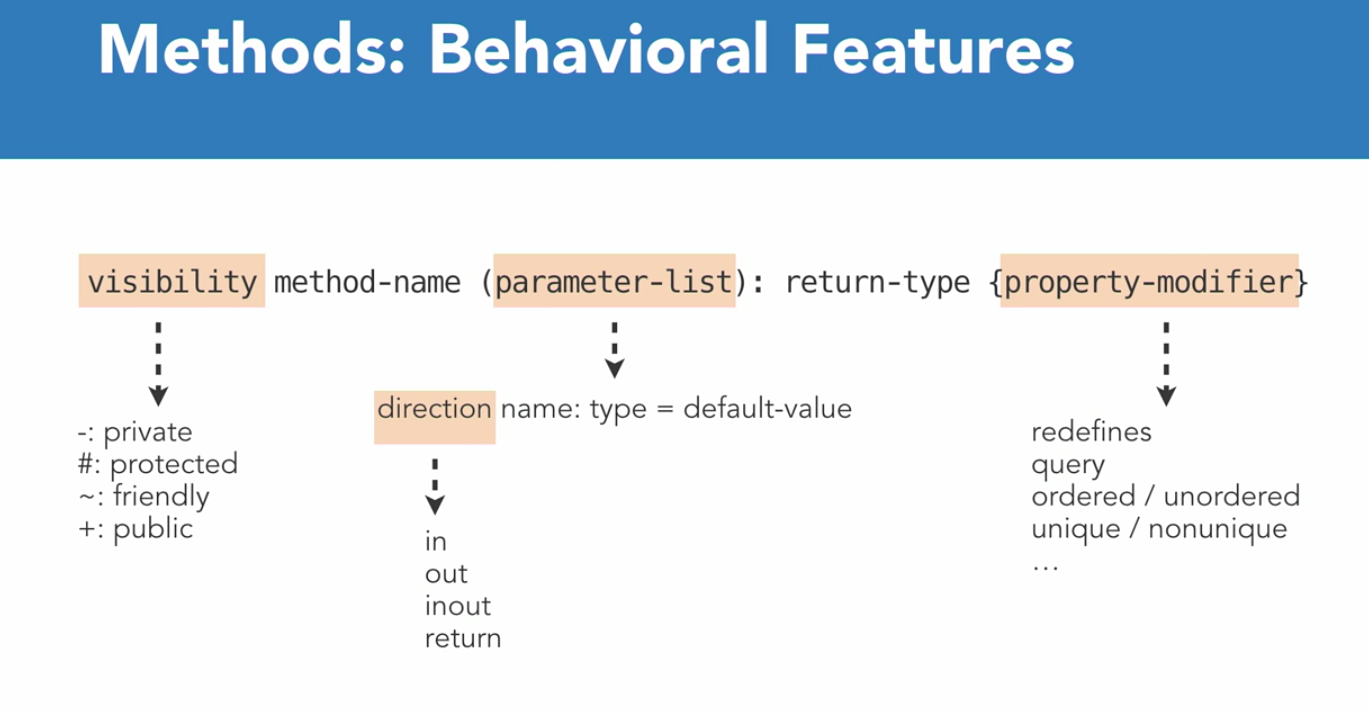
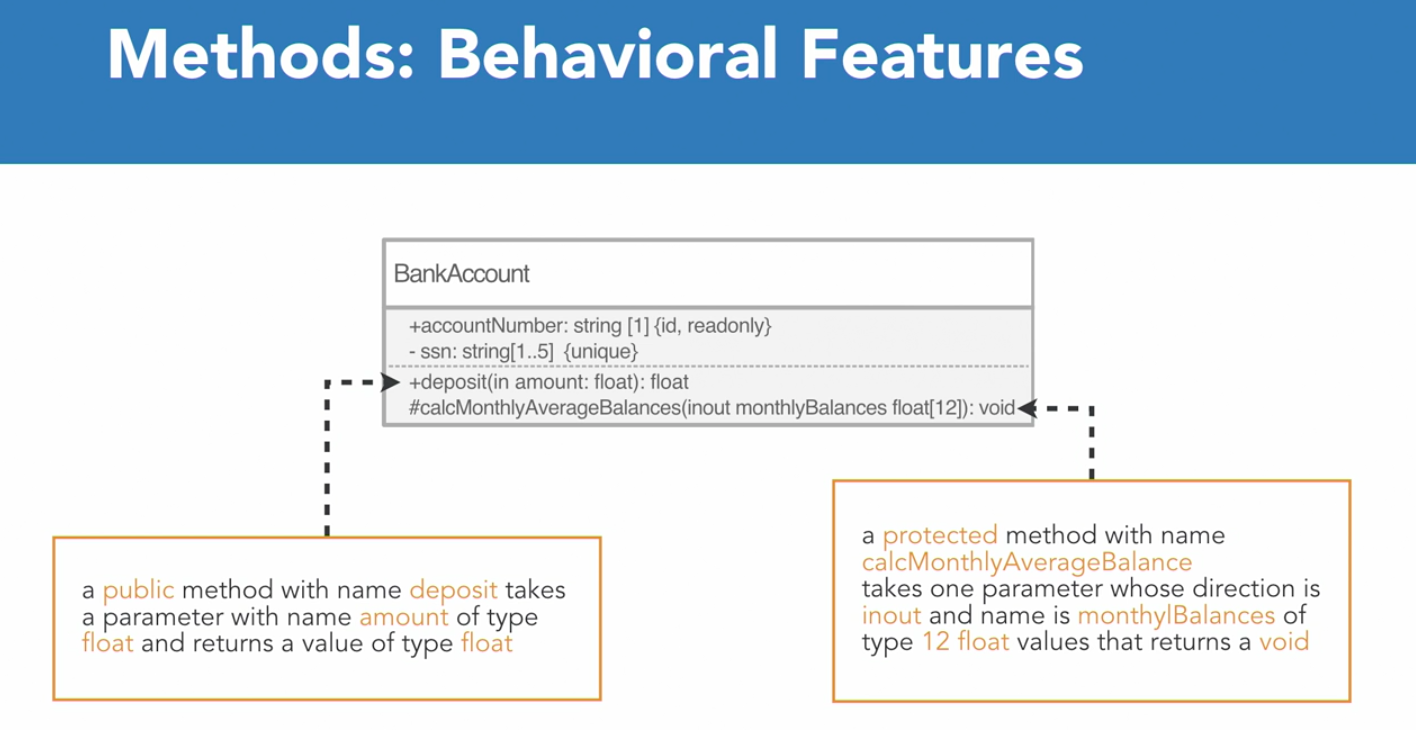
ATTRIBUTES: STRUCTURAL FEATURES



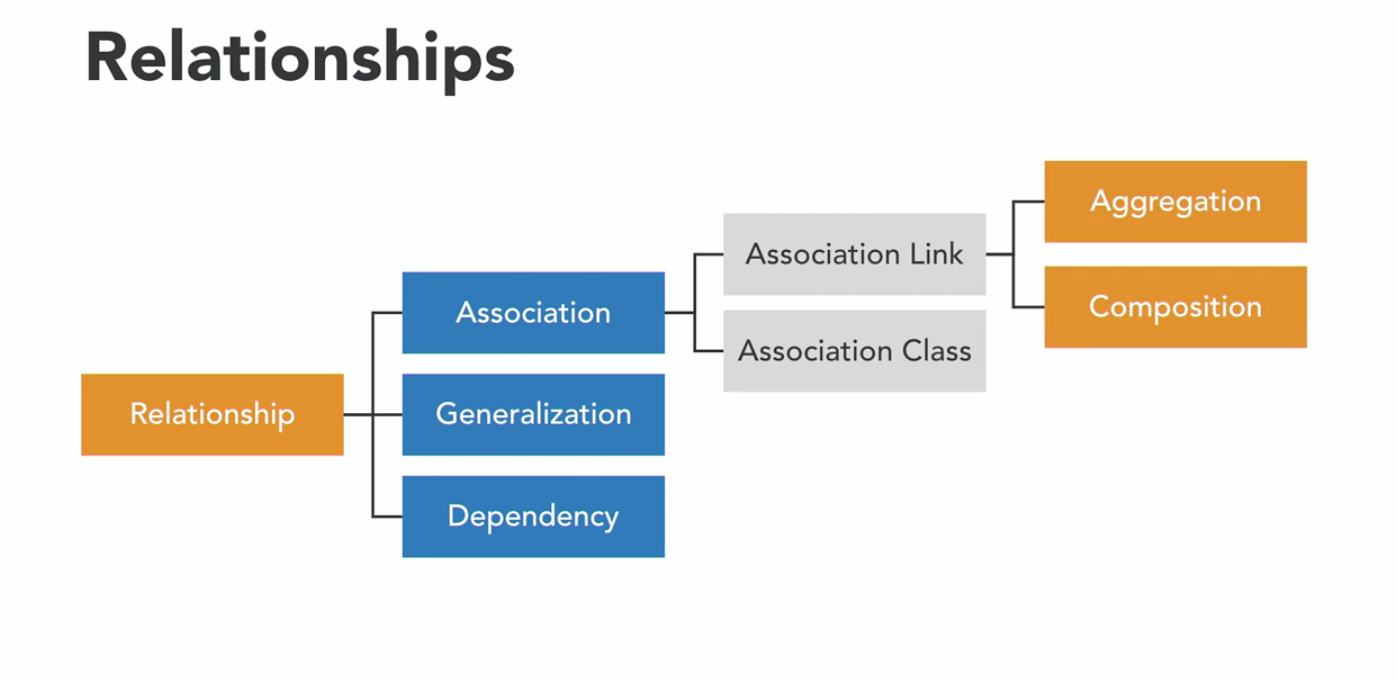


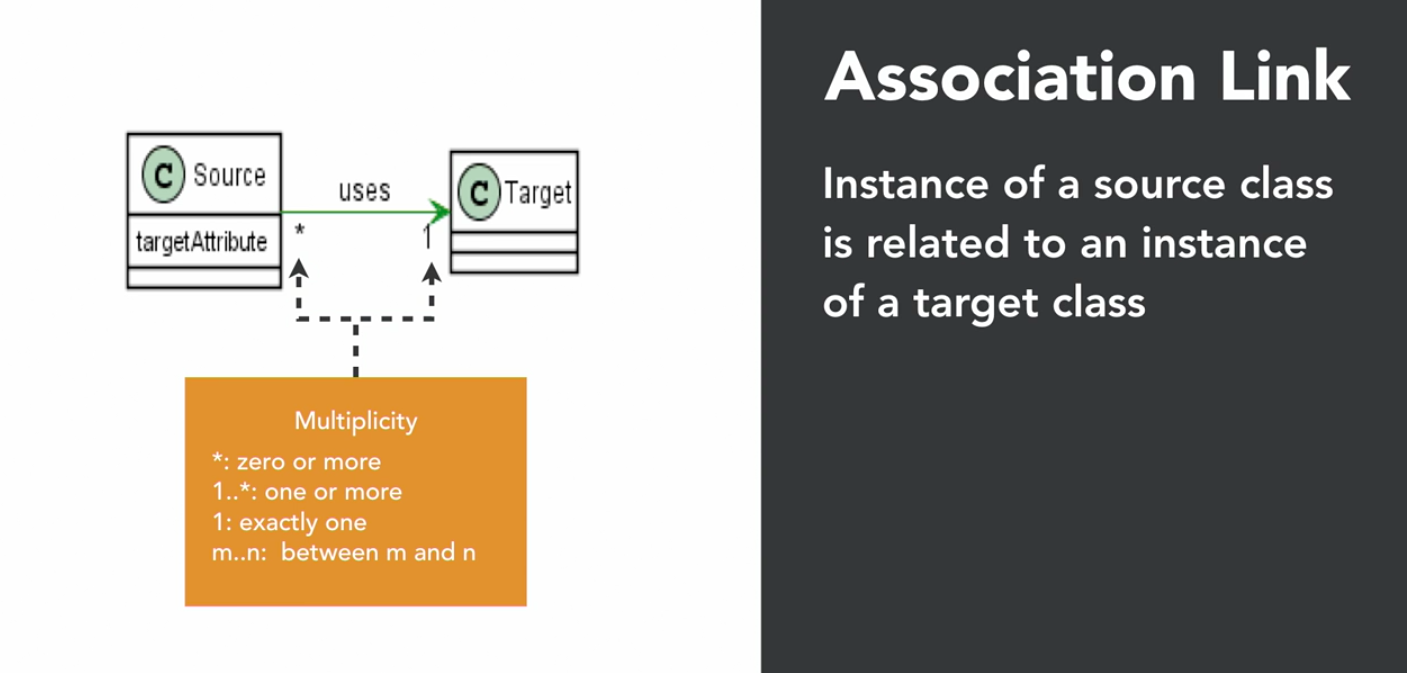


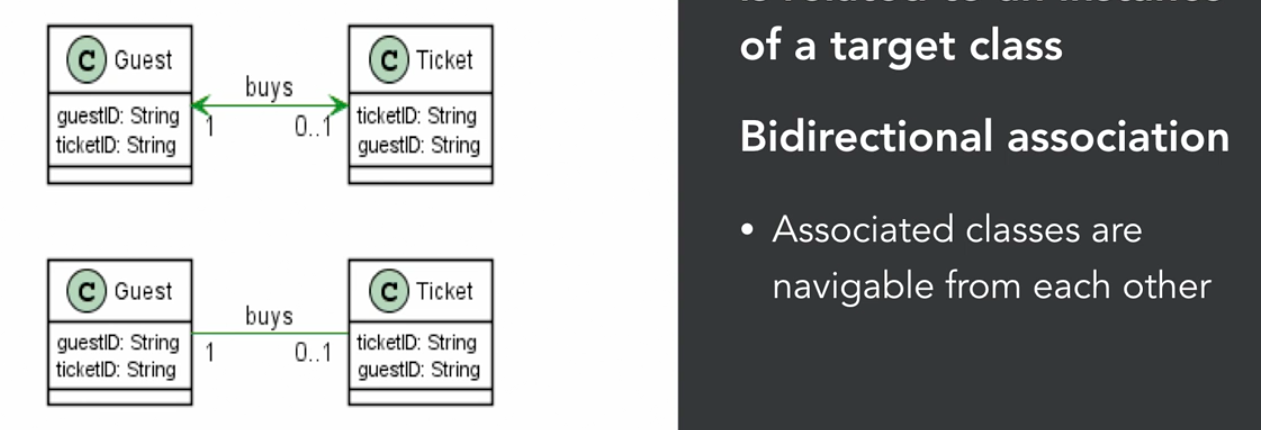
METHODS: BEHAVIORAL FEATURES

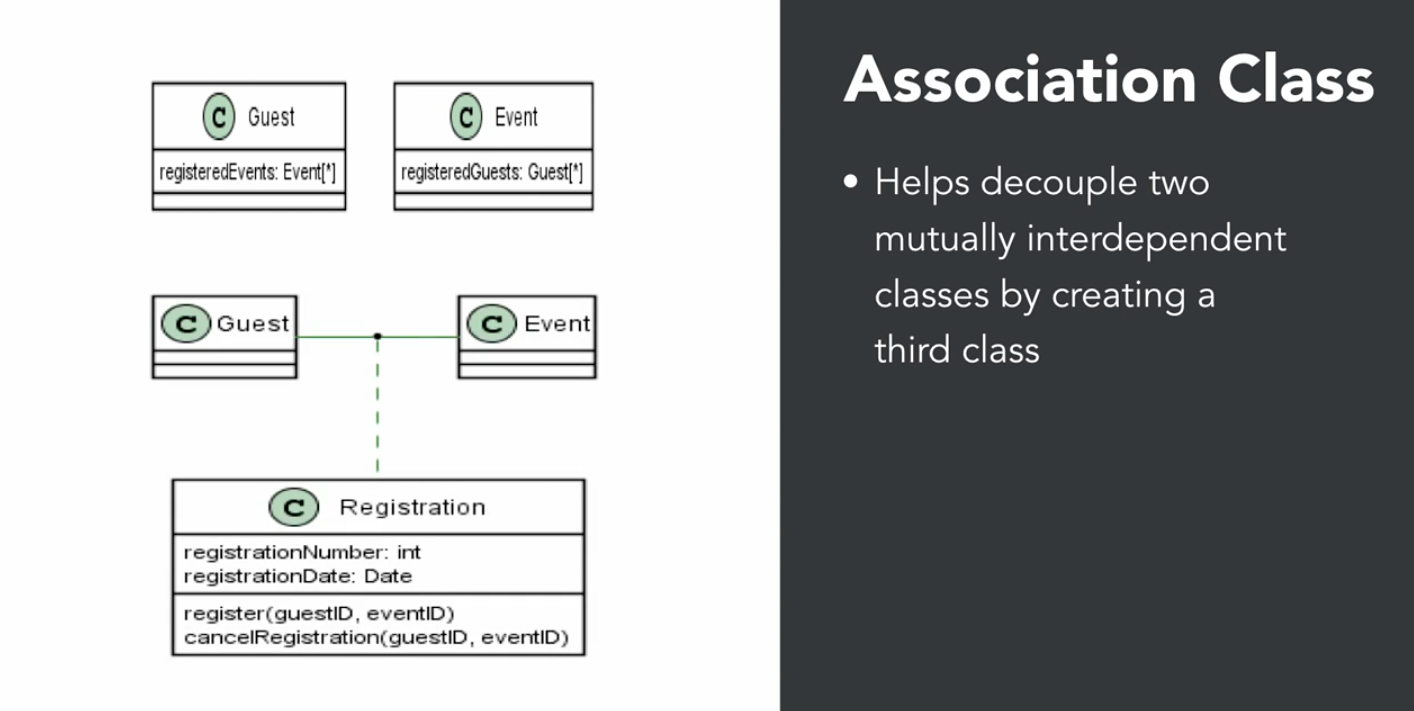


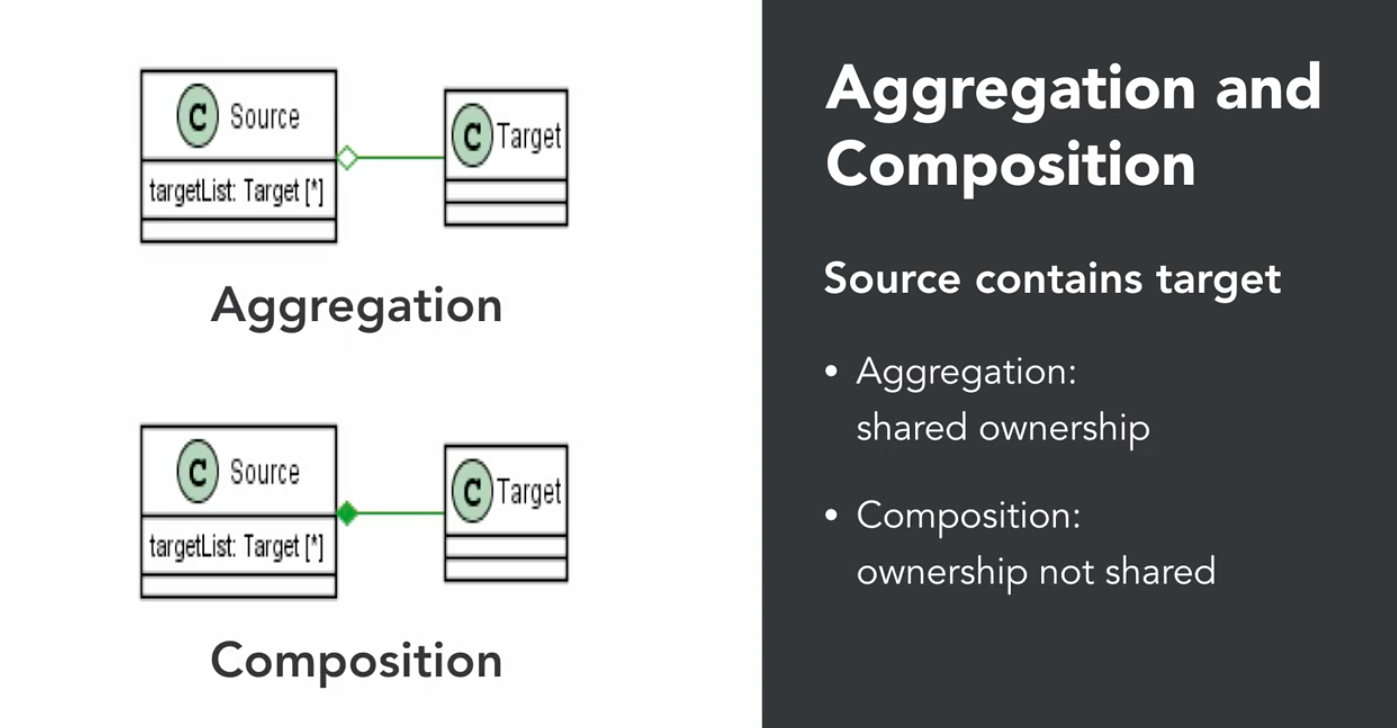
RELATIONSHIPS









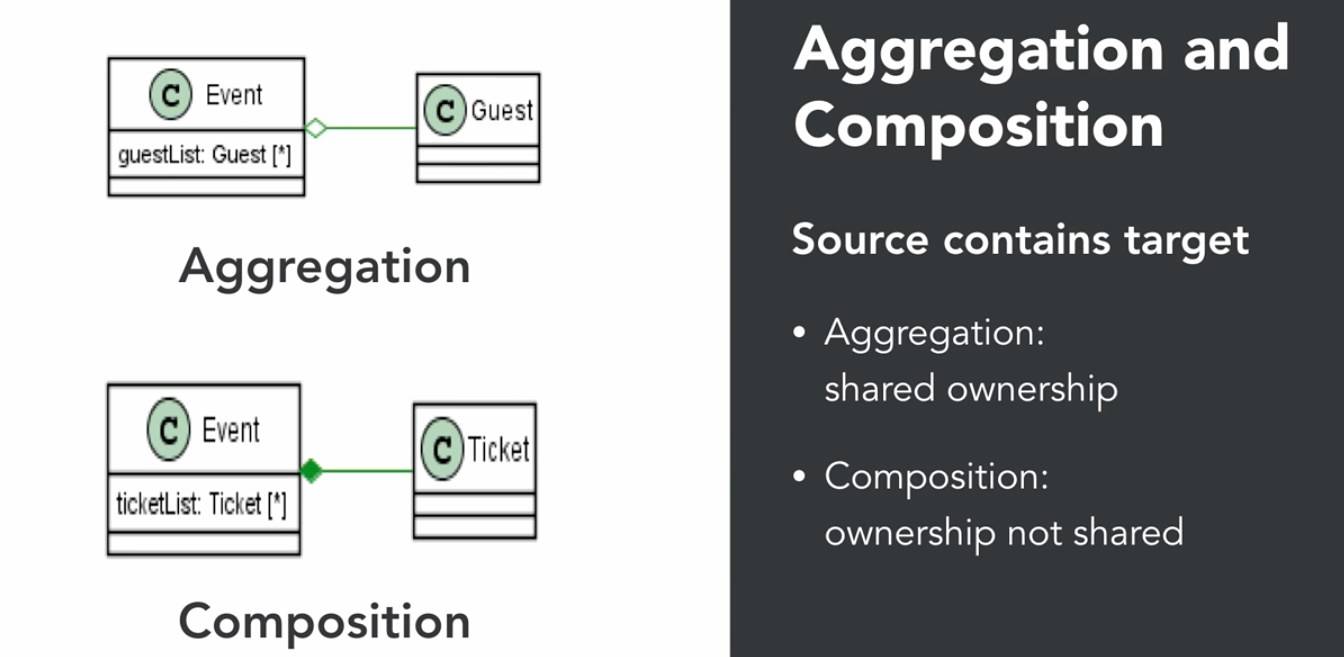


Aggregation

* “Has-a”
* Linked objects are independent of each other. Student has an address
* Child exits in the absence of parent.
* Independent existence possible
* University and Departments
* Library and Books

Composition (Stronger form of Aggregation)

* Human class is **composition** of heart and lungs
* House and Rooms (If house is demolished, the rooms do not exist)
* Computer and CPU

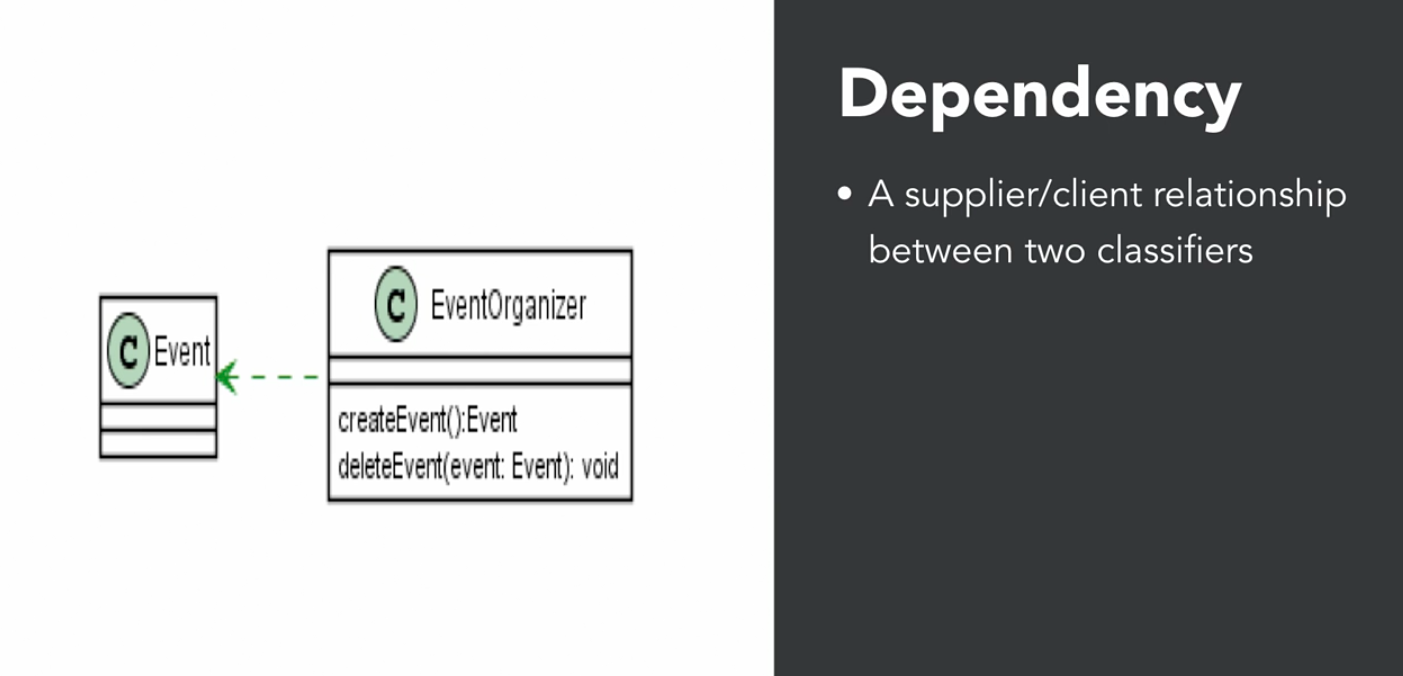
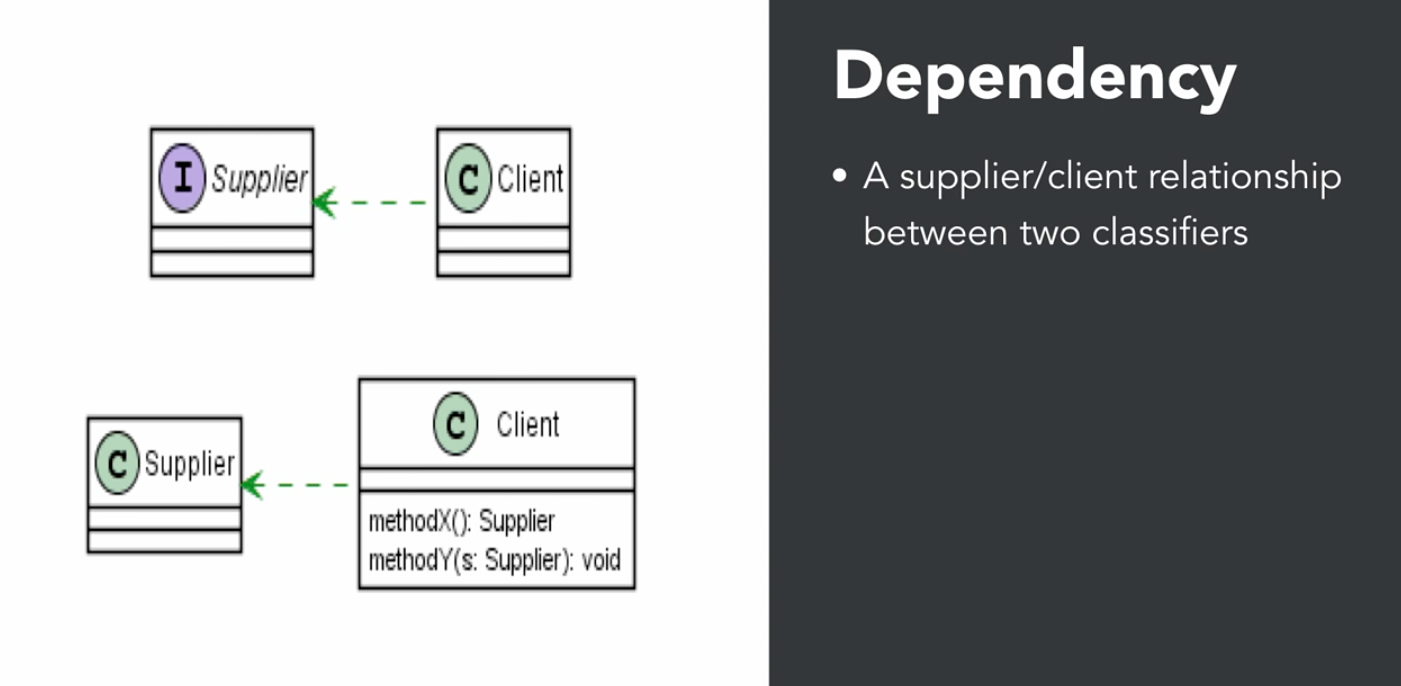


Aggregation

* Event contains target class Guest. But Guest can be a part of more than one Event
* Guest can visit other Event if this Event gets cancelled

Composition

* A ticket belongs to only one Event. If Event gets cancelled, ticket gets destroyed



Dependency because child has Event method….

Generalization (Inheritance)

