# UML Class Relationships

The following is a brief description of the relationship types used in UML class diagrams.

## Inheritance (aka generalization)

* A parent-child relationship.
* A **SubType** class inherits behavior from a **SuperType** class. A **SuperType** is a more general concept than a **SubType**. The **SuperType** is the parent class and the **SubType** is the child class.
* ***Example***: SuperType is ***Car*** and SubType is ***HybridCar***. A HybridCar is a Car. A HybridCar has all of the behaviors of a Car, and has additional behaviors not found in a Car.

## Association

* Two classes are associated with each other via object reference(s).
* First diagram shows no navigation between the two classes.

### Association with Navigation

* Second diagram shows navigation from Source to Target.
* ***Example***: Source is ***Purchase-Order*** and Target is ***Customer***. A Purchase-Order *has* a Customer that is making the purchase. The Purchase-Order class would have an attribute that stores a Customer object.
* Third diagram shows navigation in either direction.
* ***Example***: Class A is ***Purchase-Order*** and Class B is ***Customer***. A Purchase-Order *has* a Customer that is making the purchase. A Customer may have many Purchase-Orders. The Purchase-Order class would have an attribute that stores a Customer object. The Customer class would have an attribute that stores many Purchase-Order objects.

## Aggregation

* A whole-part relationship between an aggregate (the whole or container) and a constituent part (i.e., Item). Typically used in one-to-many relationships.
* Aggregation is a special type of association relationship.
* ***Example***: Container is ***Car*** and Item is ***Brake***. A Car has Brakes. The Car class would have an attribute that allows it to store many Brake objects.

## Composition

* Similar to aggregation, with an additional constraint that Item cannot exist without the Container. Like aggregation, composition is a special type of association relationship.
* ***Example***: Container is ***Invoice*** and Item is ***Purchase-Item***. An Invoice contains Purchase-Items. The Invoice class would have an attribute that allows it to store many Purchase-item objects. The Purchase-Item objects do not exist unless their Invoice object exists.

## Realizes

* Implements an interface.
* The **Class** must implement the behavior of the **Interface**.
* ***Example***: Interface is ***Door*** and Class is ***Car***. A Car must include behavior for a Door.

## Dependency

* A Client class is dependent on a Supplier class. That is, a change to the Supplier may affect or supply information needed by the Client.
* ***Example***: Client is ***Employee-GUI*** and Supplier is ***Employee***. The Employee-GUI is dependent on the Employee. When the Employee class changes, it’s likely the Employee-GUI class must be changed. When the Employee-GUI class changes, no changes are necessary in the Employee class.

# Additional UML Class Diagram Notation

The following is a brief description of other notations used in a UML class diagram.

## Association Multiplicities

|  |  |
| --- | --- |
|  | Exactly one  Many (zero or more)  Optional (zero or one) |

## Modifiers for Multivalued (\*) Multiplicity Sets

|  |  |
| --- | --- |
| {ordered}  {unordered}  {unique}  {nonunique} | The objects in the set can be retrieved in a sort order.  Default.  Each object in the set can be uniquely identified.  Default |

## Association Multiplicity Examples

|  |  |
| --- | --- |
|  | A Car is owned by zero or one Person.  A Person owns zero or more Cars.  A Person owns zero or more Cars.  A Person owns zero or more Cars, ordered by the date purchased. |

# Java Examples of UML Class Relationships

## Inheritance (aka generalization)

* ***Example***: SuperType is ***Car*** and SubType is ***HybridCar***.
* A HybridCar is a Car.

|  |  |
| --- | --- |
| public class Car  {  private String engineType;  public Car(String engineType)  {  this.engineType = engineType;  }  } | public class HybridCar **extends Car**  {  private String electricEngineSize;  public HybridCar(String engineType,  String electricEngineSize)  {  **super(engineType); //call SuperType constructor**  this.electricEngineSize = electricEngineSize;  }  } |

## Association

* Class A and Class B have a connection that involves their instances.

### Association with Navigation

* ***First Example***: Source is ***Purchase-Order*** and Target is ***Customer***.
* A Purchase-Order *has* a Customer that is making the purchase. The Purchase-Order class would have an attribute that stores a Customer object.
* ***Second Example***: Class A is ***Purchase-Order*** and Class B is ***Customer***.
* A Purchase-Order *has* a Customer that is making the purchase. A Customer may have many Purchase-Orders. The Purchase-Order class would have an attribute that stores a Customer object. The Customer class would have an attribute that stores many Purchase-Order objects.

|  |  |
| --- | --- |
| ***First Example (Source is Purchase\_Order, Target is Customer)*** | |
| public class Customer  {  //attributes and methods for a customer.  } | public class Purchase\_Order  {  Customer cust;  public Purchase\_Order(Customer cust)  {  this.cust = cust;  }  } |
| ***Second Example (Class A is Purchase\_Order, Class B is Customer)*** | |
| public class Customer  {  **//Customer may have many**  **//purchase orders.**  private ArrayList<Purchase\_Order> poList;  public Customer()  {  poList = new  ArrayList<Purchase\_Order>();  }  public void addPO(Purchase\_Order po)  {  poList.add(po);  }  } | public class Purchase\_Order  {  **//A purchase order is for one customer.**  Customer cust;  Public Purchase\_Order(Customer cust)  {  this.cust = cust;  }  } |

## Aggregation

* ***Example***: Container is ***Car*** and Item is ***Brake***.
* A Car has Brakes. The Car class would have an attribute that allows it to store many Brake objects.

|  |  |
| --- | --- |
| public class Brake  {  private String type;  public Brake(String type)  {  this.type = type;  }  } | public class Car  {  **private Brake[] brakes; //contains Brake objects**  public Car()  {  **//create array and individual Brake objects**  brakes = new Brake[2];  brakes[0] = new Brake("front disc");  brakes[1] = new Brake("rear drum");  **//Note: Many cars have disc and/or drum brakes.**  **//This is why this is aggregation.**  }  } |

## Composition

* ***Example***: Container is ***Invoice*** and Item is ***Purchase-Item***.
* An Invoice lists Purchase-Items. The Invoice class would have an attribute that allows it to store many Purchase-item objects.

|  |  |
| --- | --- |
| public class Purchase\_Item  {  private String itemDesc;  private int count;  public Purchase\_Item(String desc,  int count)  {  itemDesc = desc;  this.count = count;  }  } | public class Invoice  {  **private ArrayList<Purchase\_Item> items;**  public Invoice()  {  **//create ArrayList**  items = new ArrayList<Purchase\_Item>();  }  public addItem(String desc, int count)  {  **//create unique Purchase\_Item;**  **//store in ArrayList**  Purchase\_Item item =  new Purchase\_Item(desc, count);  items.add(item);  }  } |

## Realizes

* ***Example***: Class is ***Car*** and Interface is ***Door***.
* A Car must include behavior for a Door.

|  |  |
| --- | --- |
| public **interface** Door  {  // method signatures  public void openDoor();  public void closeDoor();  } | public class Car **implements Door**  {  private String engineType;  public Car(String engineType)  {  this.engineType = engineType;  }  **//must implement each Door method signature**  public void openDoor()  {  //code that mimics behavior of opening a door  }  public void closeDoor()  {  //code that mimics behavior of closing a door  }  } |

## Dependency

* ***Example***: Client is ***Employee-GUI***; Supplier is ***Employee***.
* The Employee-GUI is dependent on the Employee. When the Employee class changes, it’s likely the Employee-GUI class must be changed. When the Employee-GUI class changes, no changes are necessary in the Employee class.

|  |  |
| --- | --- |
| public class Employee  {  //Attributes and methods to  //represent an Employee.  } | public class EmployeeGUI  {  **//Note: no attribute of type Employee!**  **//Instead: Employee object passed to methods that**  **// need access to Employee data.**  public EmployeeGUI(Employee empl)  {  //create GUI to display empl data.  }  public void changeDisplay(Employee empl)  {  //change info displayed in GUI  //based on empl obj.  }  } |