OOP – 4 Design Principles

Abstraction

Abstraction

Abstraction is the idea of simplifying a **concept** in the problem domain to its essentials within some **context**. Abstraction allows you to better understand a concept by breaking it down into a simplified description that ignores unimportant details.

- For example, we might want to create an abstraction for a food.
- In a health context, its nutritional value and not its cost would be part of a simplified description of a food.

Abstraction

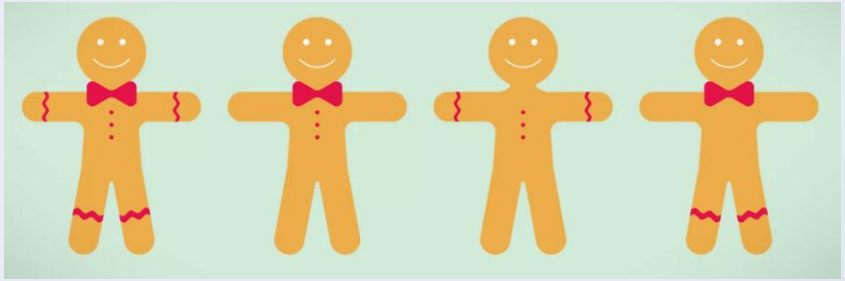
- Good abstraction emphasizes the essentials needed for the concept and removes details that are not essential.
- Also an abstraction for a concept should make sense for the concept's purpose.
- This idea applies the Rule of Least Astonishment.

Rule Of Least Astonishment

The abstraction captures the essential attributes and behavior for a concept with no surprises and no definitions that fall beyond its scope. You don't want to surprise anyone trying to understand your abstraction with irrelevant characteristics.

Abstraction

- In object oriented modeling, abstraction pertains most directly to the notion of a class.
- When you use abstraction to decide the essential characteristics for some concept, it makes the most sense to define all of those details in a class named after the concept.
- A class is like a template for instances of a concept.
- An object instantiated from a class then has the essential details to represent an instance of some concept.



Abstraction

- What are the essential characteristics of a **person** that we care about? Well, it's hard to say because **person** is so vague and we haven't said what the purpose of our person is.
- The abstractions you create are relative to some context, and there can be different abstractions for one concept.
- For example, if you are creating a **driving app**, you would care about **a person in the context of a driver**.
- In another example, if you were creating a **restaurant app**, then you would care about a person in the context of a patron.
- It is up to you to choose the abstraction that is most appropriate for your purpose.
- Before we start creating an abstraction, we need a context for our person.

Abstraction

- Context or specific perspective is critical when forming an abstraction.
- Let's look at an example where our context is an academic setting, and we want to create an abstraction for a student.

Attributes

- A house cat will have basic attributes like a **name, color, favorite nap location,** and **microchip number**.

- Certain values of these attributes could change.
- For example, over the course of the day the cat's favorite nap location could change from the living room to a bedroom.

Behaviors

- A house cat has a pretty low activity lifestyle and not much purpose other than to nap.
- Perhaps you said, **having naps, grooming,** and **catching mice in the house, eating**.
- Within the context of an abstraction, anything other than a concept's essential attributes and behaviors is irrelevant.
- When considering our student in the context of an academic setting, we don't care whether the student has a pet or how they clean their kitchen or what their favorite video game is; Those are all irrelevant details to the abstraction in this context.

Abstraction

- Whenever we make abstractions, we need to remember our context.
- If the context changes, the right abstraction can as well.
- Say our context changes and we need to model a student from a social perspective.
- How would our definition change?
- Perhaps the relevant attributes would be the student groups they belong to, their hobbies for study breaks, and the sports teams their in.

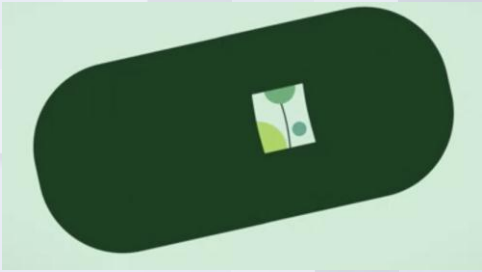
Encapsulation

Encapsulation

- There are many things that you can represent as objects.
- For example, you could represent a **university course** as an **object**. The **course object** can have many **attribute values**, like the specific number of students enrolled, credit value, and prerequisites, as well as specific **behaviors** dealing with these values and the course class defines the essential attributes and behaviors of all the course objects.
- Encapsulation involves three ideas. As the name suggests, it's about making a sort of capsule.
- The capsule contains something inside, some of which you can access from the outside, and some of which you cannot.

Encapsulation

- First, you ***bundle*** **attribute values** or **data**, and **behaviors** or **functions**, that manipulate those values together into a self-contained object.
- Second, you can ***expose*** certain data and functions of that object, which can be accessed from other objects.
- Third, you can ***restrict*** access to certain data and functions to only within that object.



So..

Encapsulation

Encapsulation forms a self-contained object by bundling the data and functions it requires to work, exposes an interface whereby other objects can access and use it, and restricts access to certain inside details.

- You naturally bundle when you define a class for a type of object.
- Abstraction helps to determine what attributes and behaviors are relevant about a concept in some context.

Encapsulation

- The distinct objects thus made from a particular class, will have their own data values for the attributes and exhibit resulting behaviors.
- You'll find that programming is easier when the data, and the code that manipulates that data, are located in the same place.

Encapsulation

- An object's data should only contain what is **relevant** for that object.
- A **student** would only contain relevant data for themselves, like the degree program.
- This is, as if the **student object** knows its degree program like a **real student** would.
- A **course object** would know a list of students taking it.
- The **professor object** would know a list of courses the professor teaches and none of these types of objects would contain a list of other courses offered, because that is not relevant data for them.

Encapsulation

- Besides **attributes**, a class also defines **behaviors** through **methods**.
- For an object of the class, the methods manipulate the attribute values or data in the object to achieve the **actual behaviors**.
- You can **expose** certain **methods** to be accessible to objects of other classes, thus, providing an **interface** to use the class.
- For example, a course can provide a **method** to allow a student to **enroll** in the course, or a course that a professor is teaching, can provide a method that allows the professor to **see** the list of students in that course.

Encapsulation

- Encapsulation helps with **data integrity**. You can define certain attributes and methods of a class to be **restricted** from outside to access.
- In practice, you often present outside access to all the attributes, except through specific methods.
- That way, the attribute values of an object cannot be changed directly through variable assignments, Otherwise, such changes could break some assumption, or dependency for the data within an object.

Encapsulation

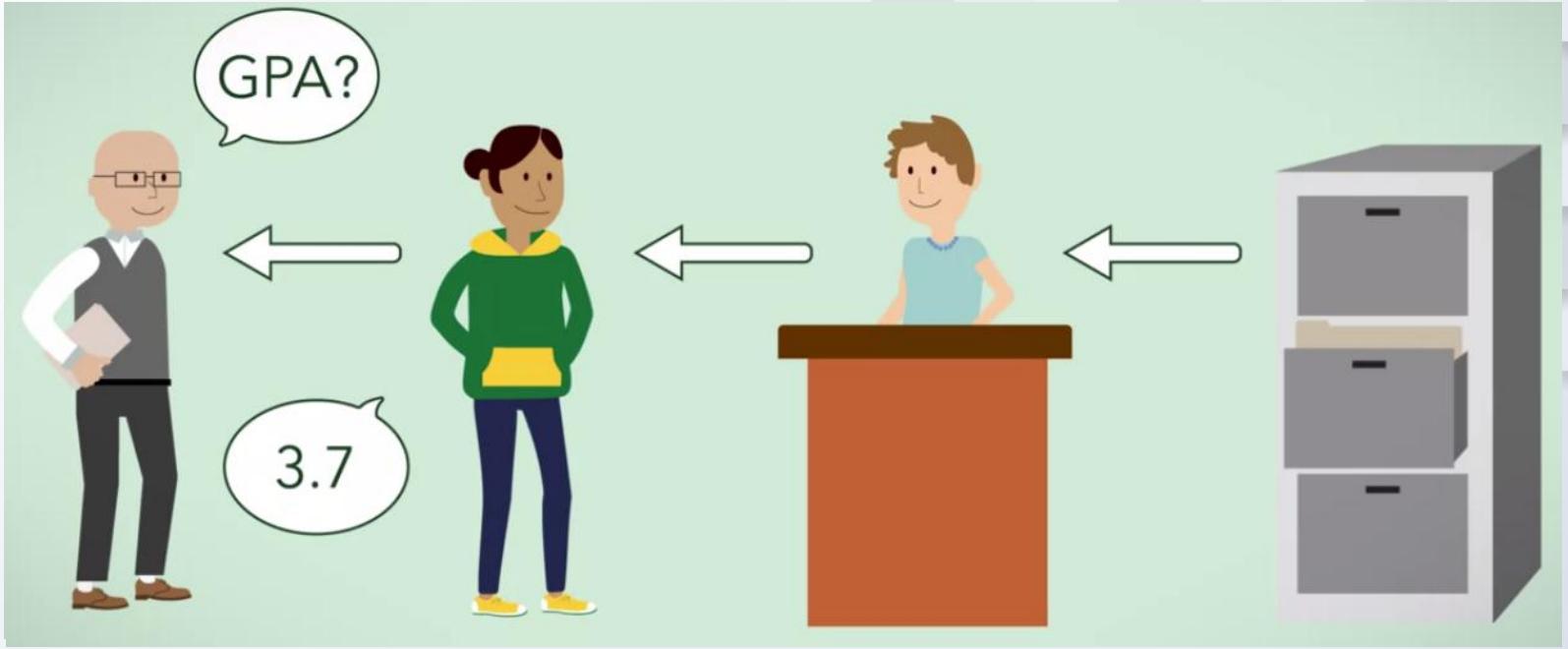
- As well, Encapsulation can secure sensitive information.
- For example, you may allow a student class to store a degree program and grade point average, GPA.
- The student class itself could support queries involving the GPA, without necessarily revealing the actual value of the GPA.
- For example, the student class could provide a method that tells whether the student is in good standing for the degree program, which uses the GPA and the calculation, but never reveals its actual value.

Encapsulation

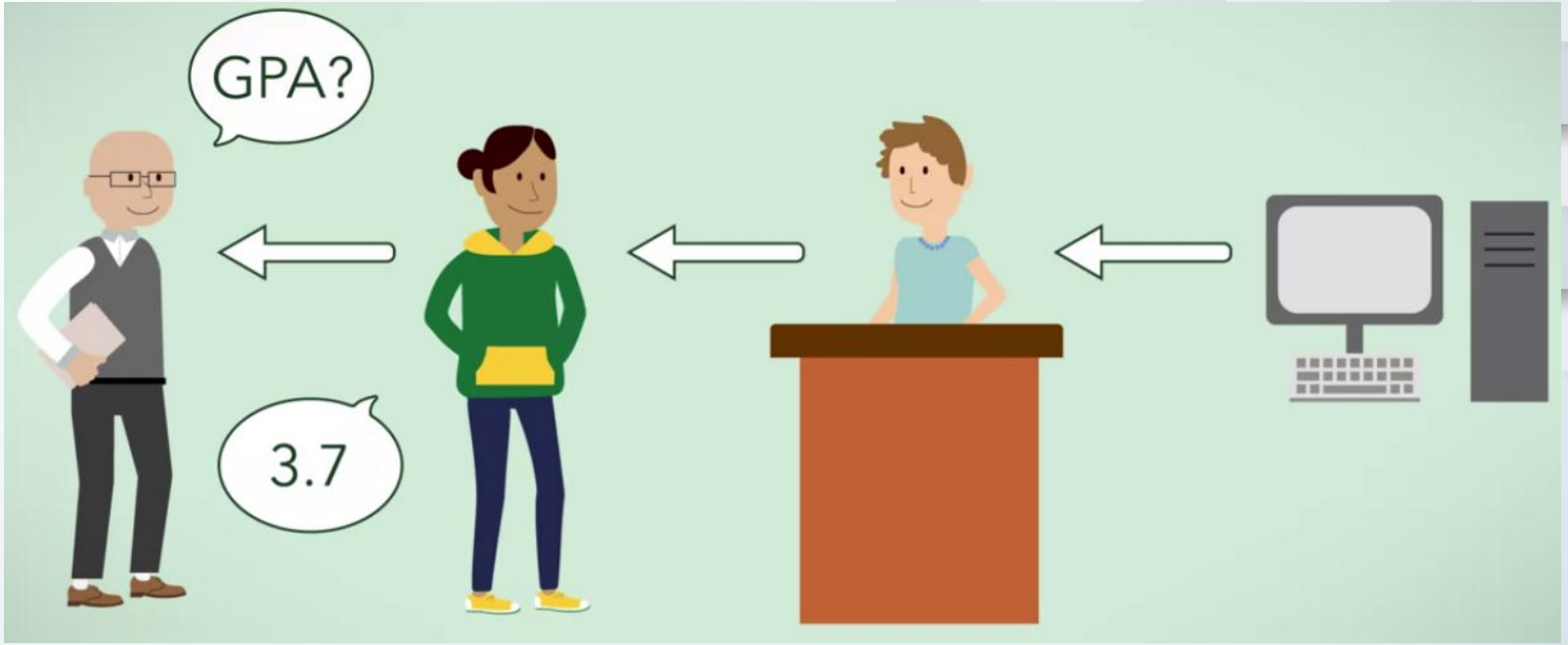
- Encapsulation helps with **software changes**.
- The accessible interface of a class can remain the same, while the implementation of the attributes and methods can change.
- Outsiders using the class, do not need to care how the implementation actually works behind the interface.

Encapsulation

- Let's say, a professor wants a student to calculate and declare their GPA before getting admitted into a course.
- This is an action that the student may perform, however, there could be many different ways that this action could be done.
- A student might fill out a paper form and hand that to the administrator, who would check the paper files, write it a list of courses and grades for that student, from which to calculate the GPA.



- This action may have been automated, so that the student would fill out the paper form and hand it into an administrator, who would check the computer database for the GPA **or** the student can go online, and get the GPA themselves using a student information system.

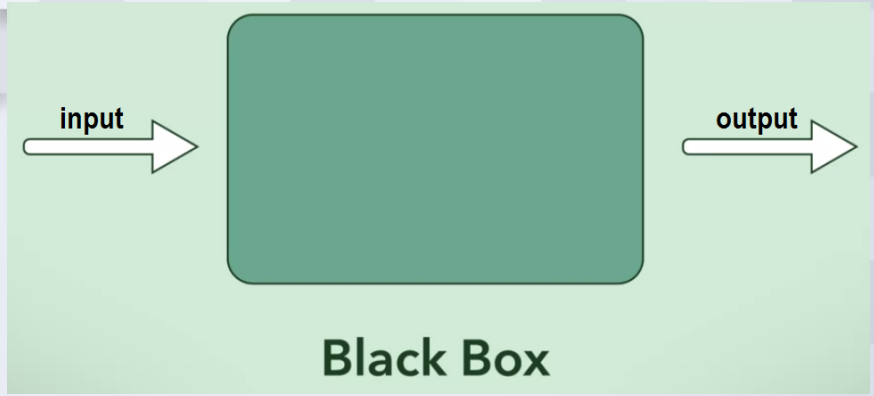


- As you can see, the professor does not have to care how the student gets their GPA, a student is the only one that really needs to know how to do it.

Encapsulation

- You can apply this notion when you program software.
- For example, suppose a student class has a method to return its major when called.
- The actual steps to retrieve the major does not need to be known by any other class.
- In programming, this sort of thinking is commonly referred to as, **Black Box Thinking**.

Encapsulation



- Think of a class like a black box that you cannot see inside for details about, how attributes are represented, or how methods compute the result, but you provide inputs and obtain outputs by calling methods.
- It doesn't matter what happens in the box to achieve the expected behaviors.
- This distinction between what the outside world sees of a class, and how it works internally is important.

Encapsulation

- Encapsulation achieves what is called, the **Abstraction Barrier**.
- Since the internal workings are not relevant to the outside world, this achieves an abstraction that effectively reduces complexity for the users of a class.
- This **increases re-usability**, because another class only needs to know the right method to call to get the desired behavior, what arguments to supply as inputs, and what appear as outputs or effects.

Encapsulation

- In the real world, if I ask you to buy me a **soda**, you can get the soda in many ways.
- You can go to the **vending machine** and buy one, **or** you can drive to another city and buy one there.
- **The input and output is the same**.
- I ask you to buy me a soda, and you give me a soda.
- I don't need to know the details of how you got it.

So..

- **Encapsulation** is a key design principle in achieving a well written program.
- It keeps your software modular and easier to work with.
- It also keeps your classes easy to manage, whose behaviors are accessed like black boxes.

Decomposition

Decomposition

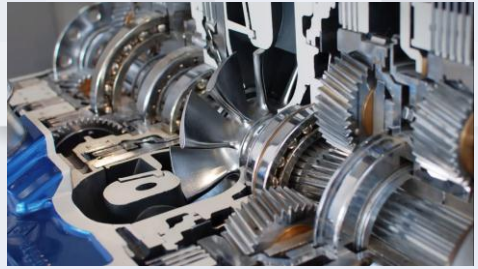
Decomposition is taking a **whole** thing and dividing it up into different **parts**. Or, on the flip side, taking a bunch of separate parts with different functionalities and combining them together to form a whole. Decomposition allows you to further break down problems into pieces that are easier to understand and solve.

- Let's consider a whole thing, like a car or a refrigerator.
- By breaking down such a thing into its different parts using decomposition, you can more easily keep their different responsibilities separate.

Decomposition

- A general **rule for decomposition** is to look at the different **responsibilities** of some whole thing, and evaluate how you can separate them into different parts, each with its own specific responsibility.
- This relates one whole to multiple different parts.
- Let's get some practice breaking down a whole thing into its different constituent parts.
- **Identify the different constituent parts of a car.**

Decomposition



- Some possible parts of your car maybe a **transmission**, a **motor**, **wheels**, tires, doors, windows, seats, and fuel.
- For another example, a refrigerator also has several parts.
- There is the cabinet and doors, compressor and coils, freezer, ice-maker, shelves, drawers, and of course food if the refrigerator is stocked.
- Each of those parts has a very specific purpose to help achieve the responsibilities of the whole.
- For refrigerator, the parts work together to achieve the overall purpose of keeping, and preserving food in cold storage.

Decomposition

- Sometimes the whole will delegate specific responsibilities to the parts. So, the refrigerator delegates the freezing of food and the storing of that food to the freezer.
- Since decomposition allows you to create clearly defined parts, it is quite natural that these parts are separate.
- Let's see how parts interact within the whole.

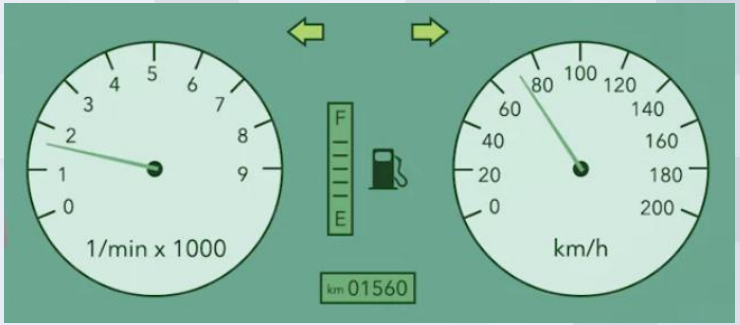
Decomposition

- A **whole** might have a **fixed** or **dynamic** number of a certain type of part.
- If there is a fixed number, then over the lifetime of the whole object it will have exactly that much of the part object.
- For example, a refrigerator has a fixed number of freezers, just one.
- This does not change over time, but there are sometimes parts with a **dynamic** number.
- Meaning, the whole object may gain new instances of those part objects over its lifetime.
- For example, a refrigerator can have a dynamic number of shelves or food items over time.

Decomposition

- Fixed parts don't change in number over time.
- So by a process of elimination, we can find out which car parts are dynamic. We know a typical car has one steering wheel, four tires, and one engine at any given time, These numbers do not fluctuate.
- The number of passengers can change though, making it dynamic.
- A **part** itself can also serve as a **whole** containing further constituent parts (**parts containing parts**).
- When you look at our refrigerator example, a drawer can contain fruit.
- Let's come up with another example of one part containing other parts.

Decomposition



- Thinking of a car, can you describe an example of decomposition where one part contains another?
- You may have come up with several different examples.
- The one that comes to mind for me is the instrument panel, which contains a fuel gauge, an odometer, and a speedometer among other parts.
- Also, because of **encapsulation**, the instrument panel treats the instruments like **black boxes**, and doesn't care about how they are implemented.

Decomposition

- One issue in decomposition involves the **lifetimes of the whole object**, and the part objects, and how they could relate. Lifetimes might be closely related.
- For example, the refrigerator and its freezer have the same lifetime, one cannot exist by itself without the other.
- If you dispose off the refrigerator, you would dispose off the freezer as well.
- But lifetime can also not be so related.
- The refrigerator and food items have different lifetimes.
- Either can exist independently.

Decomposition

- Let's take a look at the lifetimes of parts in an example.
- Consider the lifetime of a car, and name one part of a car that has a closely related lifetime, and one part that is not.
- As possible responses, a closely related lifetime would be the frame, and not closely related would be the tires.

Decomposition

- You can have whole things contain parts that are **shared** among them at the same time.
- How can this relationship arise? Consider a person who has a daughter in one family, but also a spouse in another family.
- The two families are regarded as separate wholes, but they simultaneously share the same part.
- However, sometimes sharing is not possible or intended.
- For example, a food item in a refrigerator cannot at the same time also be inside an oven.

Decomposition

- Overall, decomposition helps you to break down a problem into smaller pieces.
- A complicated whole thing can be composed out of constituent, separate, simpler parts. Important issues to understand are how the parts relate to the whole, such as **fixed or dynamic number**, their **lifetimes**, and whether there is **sharing**.

Generalization

Generalization

- The idea behind object-oriented modeling and programming is to create computer representations of concepts in the **problem space**.
- It lets you model the relative attributes in behaviors so that a computer can simulate them.
- One design principle called **generalization**, helps us to **reduce the amount of redundancy** when solving problems.
- You probably already have experience with a form of generalization and don't even know it.
- Many behaviors and systems in the real world operate through **repetitious actions**.

Generalization

- We can model behaviors using methods.
- It lets us generalize behaviors and it eliminates the need to have identical code written throughout a program.
- Take this array creation and initialization code for instance.

```
public void createArrays() {
     int[] array1 = new int[3];
     for (int i = 0; i < array1.length; i++)
          array1[i] = i;

     int[] array2 = new int[6];
     for (int i = 0; i < array2.length; i++)
          array2[i] = i;
}
```

```
public void createArrays() {
     int[] array1 = initArray(3);
     int[] array2 = initArray(6);
}

public int[] initArray(int n) {
     int[] arr = new int[n];
     for (int i = 0; i < n; i++)
          arr[i] = i;
     return arr;
}
```

Generalization

- We can generalize repetitious code that we would need to write by making a separate method and calling it.
- This helps us to reduce the amount of near identical looking code throughout our system.

Generalization

- Methods are a way of applying the same behavior to a different set of data.
- Generalization is frequently used when designing algorithms, which are meant to be used to perform the same action on different sets of data.
- We can generalize the actions into its own method, and simply pass it through a different set of data through arguments.

Generalization

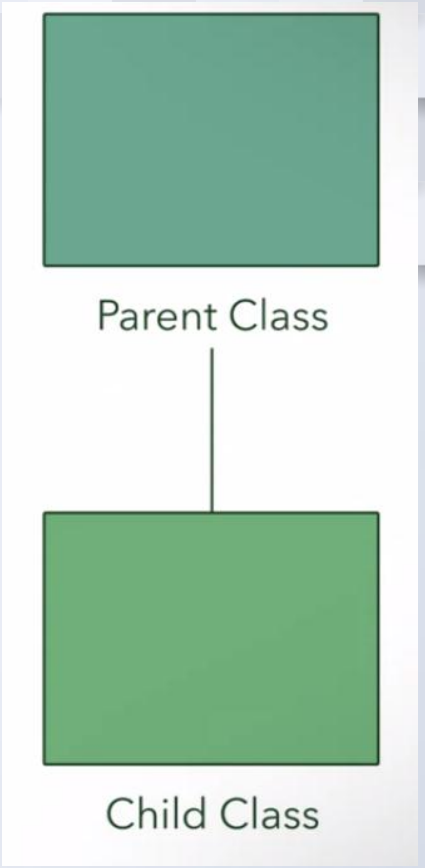
- So where else can we apply generalization?
- Well, if we can reuse code that is inside a method and a method is inside a class, then can we reuse code from a class?
- Can we generalize classes?
- Generalization happens to be one of the main design principles of object-oriented modeling and programming.
- But it's achieved differently than what we've just seen with methods.
- So how is this done?

Generalization

- Generalization can be achieved by classes through inheritance.
- In generalization we take **repeated, common, or shared characteristics between two or more classes** and factor them out into another class.

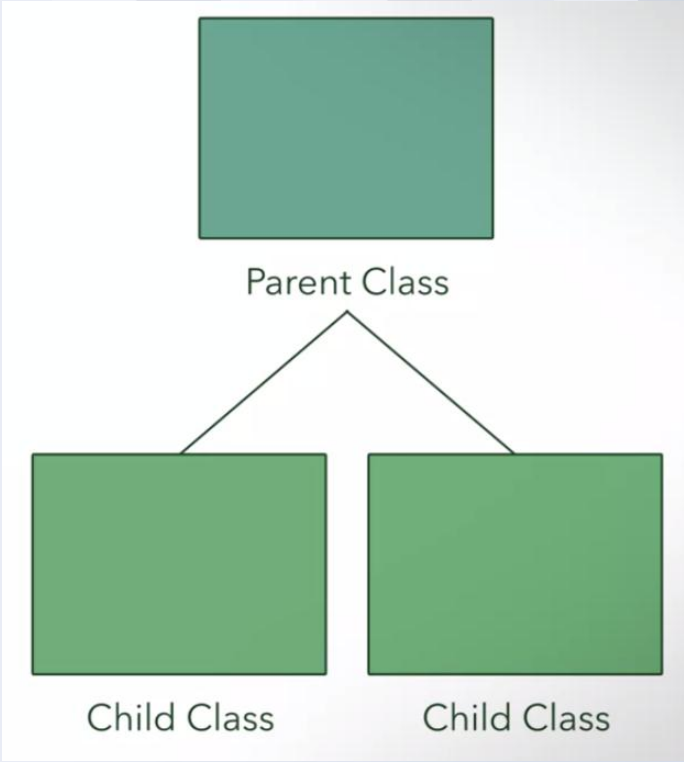
Generalization

- Specifically, you can have two classes, a parent class and a child class.
- When a child class inherits from a parent class, the child class will have the attributes and behaviors of the parent class.
- You place common attribute and behaviors in your parent class.



Generalization

- There can be multiple child classes that inherit from a parent class, and they all will receive these common attributes and behaviors.
- The child classes can also have additional attributes and behaviors, which allow them to be more specialized in what they can do.
- In standard terminology, a parent class is known as a superclass and a child class is called the subclass.



Generalization

- Let's say you want to model an adorable **cat** named **Mittens**.
- Mittens has **four legs**, **a tail**, knows how to **walk, run** and **eat**, but these attributes and behaviors can also be used to describe **Doug** the **dog.**
- Doug also has **four legs**, **a tail**, can **walk, run** and **eat.**
- If you were to design classes, cat and dog, based on these characteristics, you would have **a lot of overlapping code**.
- What would happen if you want to model another similar characteristic?

Generalization

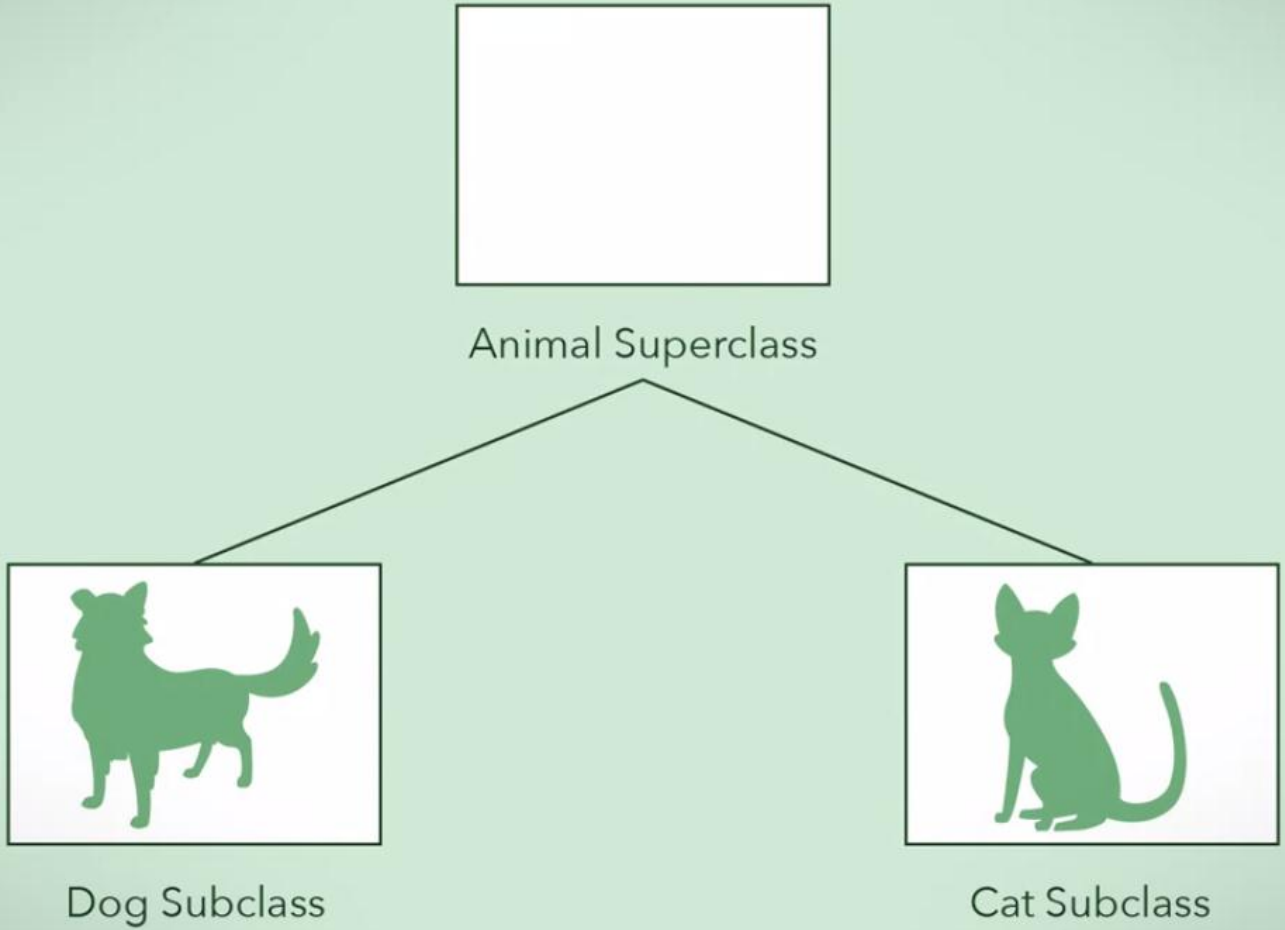
- You would need to add the same code to both of your classes.
- This doesn't sound too bad because you only have two classes, but **what if** you have several classes that share common characteristics?
- It would be time consuming and error-prone to carefully add code to all of them.
- That leads to a system that is not flexible, maintainable, or reusable.

Generalization

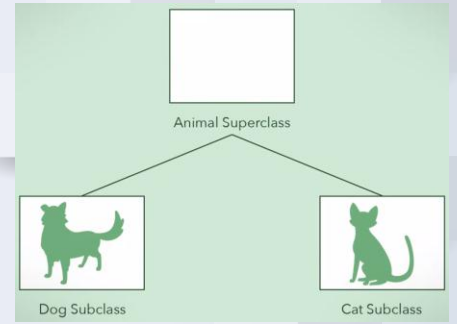
- Let's look at **Mittens** and **Doug** to see what commonalities we can find between them.
- I would say they are both a type of animal with legs, a tail and have a shared set of behaviors like, walking, running and eating.
- An animal, then, is a **general** idea.
- This means that an animal is a broad term used to describe a large grouping of more distinct classes.
- The term animal can be used to define **a set of common characteristics and behaviors** that belong to different specific types of animals, like cat and dog.

Generalization

- Let's look at **Mittens** and **Doug** to see what commonalities we can find between them.
- I would say they are both a type of animal with legs, a tail and have a shared set of behaviors like, walking, running and eating.
- An animal, then, is a **general** idea.
- This means that an animal is a broad term used to describe a large grouping of more distinct classes.
- The term animal can be used to define **a set of common characteristics and behaviors** that belong to different specific types of animals, like cat and dog.


Animal Superclass
Dog Subclass
Cat Subclass

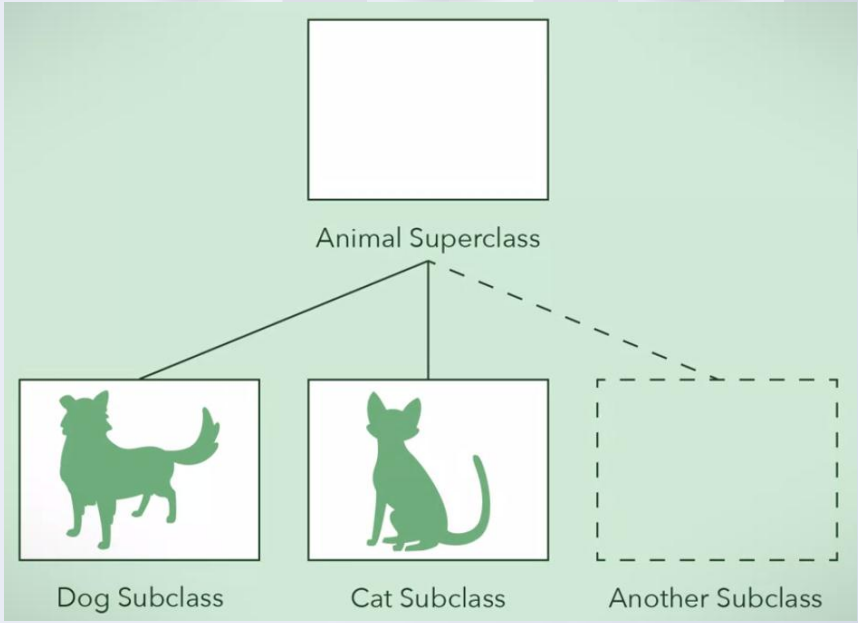
Generalization



- In this example, we can generalize the common attributes and behaviors of the cat and dog class into a superclass that we will call animal.
- Keep in mind that we can name these classes whatever we want, but since we're creating meaningful obstructions of things in the real world, our classes should be named after the things we are trying to model, this makes our code easier to understand.
- The subclasses will inherit attributes and behaviors from the superclass.
- Since cats and dogs are both animals, they inherit from the animal superclass.

Generalization

- One of the advantages of doing this is that any changes to the code that is common to both subclasses, can be made in just a superclass.
- The second benefit is that we can easily add more animals to our system, without having to write out all the common attributes and behaviors for them.
- Through inheritance, all subclasses of the animal class will be endowed with the animal classes attributes and behaviors.



Generalization

- Inheritance and methods exemplify the generalization design principle.
- There are techniques that left us apply a rule called **D.R.Y.**, which stands for **Don't Repeat Yourself**.
- We can write programs that are capable of performing the same tasks but with less code.
- It makes code more reusable because different classes or methods can share the same blocks of code.
- Systems become easier to maintain because we do not have repetitious code.
- Generalization will help you build software that is easier to expand, easier to apply changes to and easier to maintain. By learning how to identify commonalities between classes and their behaviors, you can design highly robust software solutions.

Abstraction in Java and UML

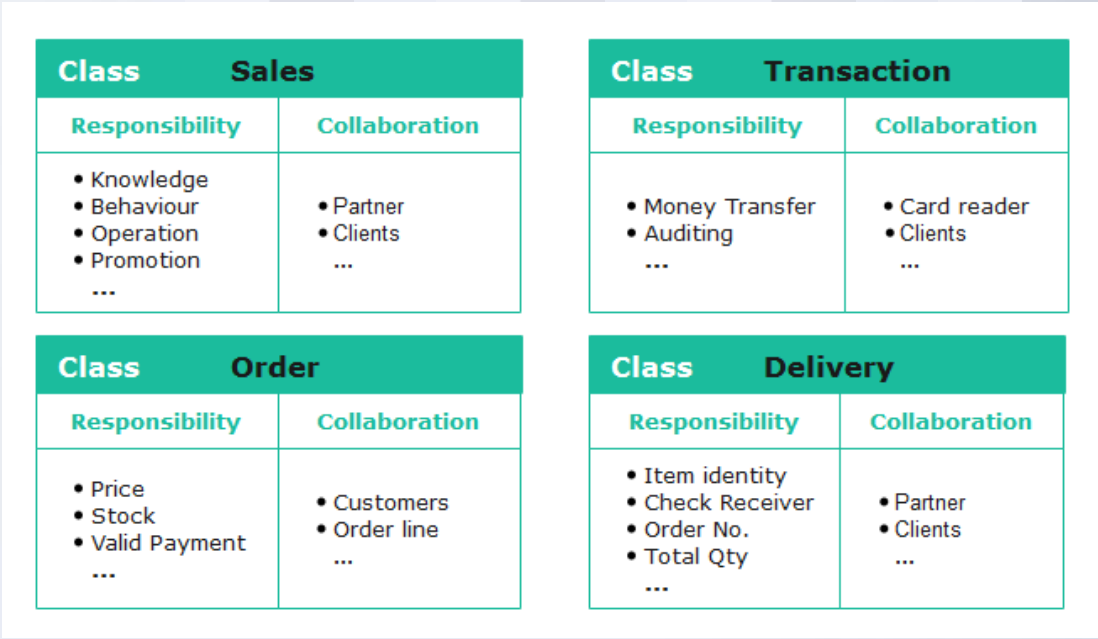
- When designing a building, architects create sketches to visualize and experiment with various designs.
- Sketches are quick to produce and an intuitive way to communicate the design to their client but these sketches are simply not detailed enough for the builders.
- When architects communicate with the people who will be constructing the building, they provide detailed blueprints which contain exact measurements of various components. These extra details allow the builders to construct exactly what the architect envisions.

Abstraction in Java and UML

- For software, developers use technical diagrams called UML Diagrams to express their designs.
- Class Diagrams are much closer to the implementation and can easily be converted to classes in code.
- **Abstraction**, which you may recall, is the idea of simplifying a concept in the problem domain to its essentials within some context.
- Abstraction allows you to better understand a concept by breaking it down into a simplified description that **ignores unimportant details**.
- You can first apply abstraction at the design level using UML Class Diagrams then eventually convert the design into code.

Abstraction in Java and UML

- You are going to learn how to abstract concepts as a class in a Class Diagram.
- You will see that additional details can be represented in a Class Diagram compared to a CRC card.
- CRC cards still have their place in prototyping and simulating various designs, but UML Class Diagrams are more suited for communicating the technical design of the software's implementation.



CRC Cards

- **Class-responsibility-collaboration (CRC)** cards are a brainstorming tool used in the design of object-oriented software.
- They were originally proposed by Ward Cunningham and Kent Beck as a teaching tool but are also popular among expert designers and recommended by extreme programming practitioners.

CRC Card for Food

- Think for a moment on how you would abstract a food item in the context of a grocery store using a CRC card.
- You would have a food component with the responsibility of keeping track of its **grocery ID, name, manufacturer, expiry date** and **price.**
- It also needs to know if the item is on sale or not.

CRC Card for Food

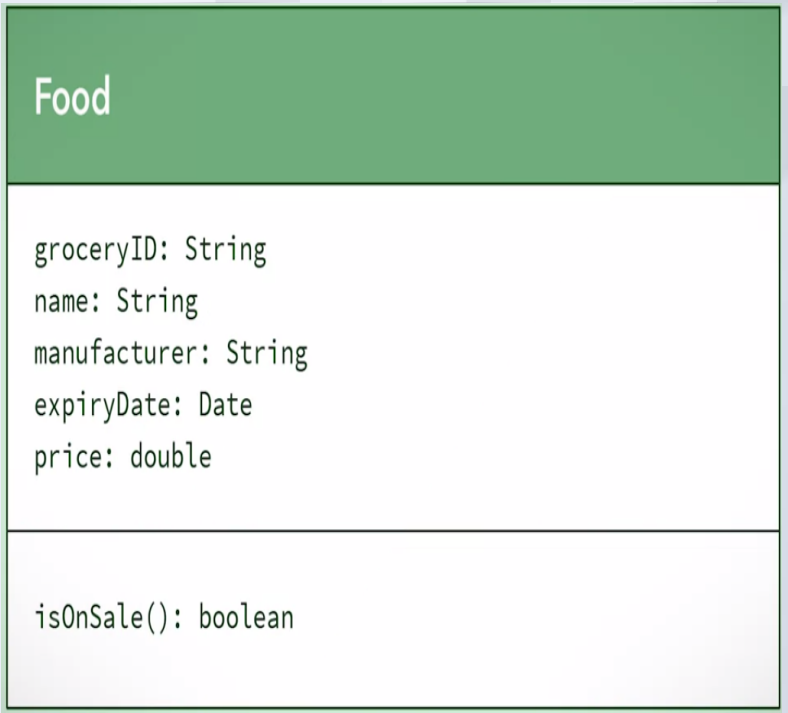
Food	
Know grocery ID Know name Know manufacturer Know expiry date Know price Check if on sale	

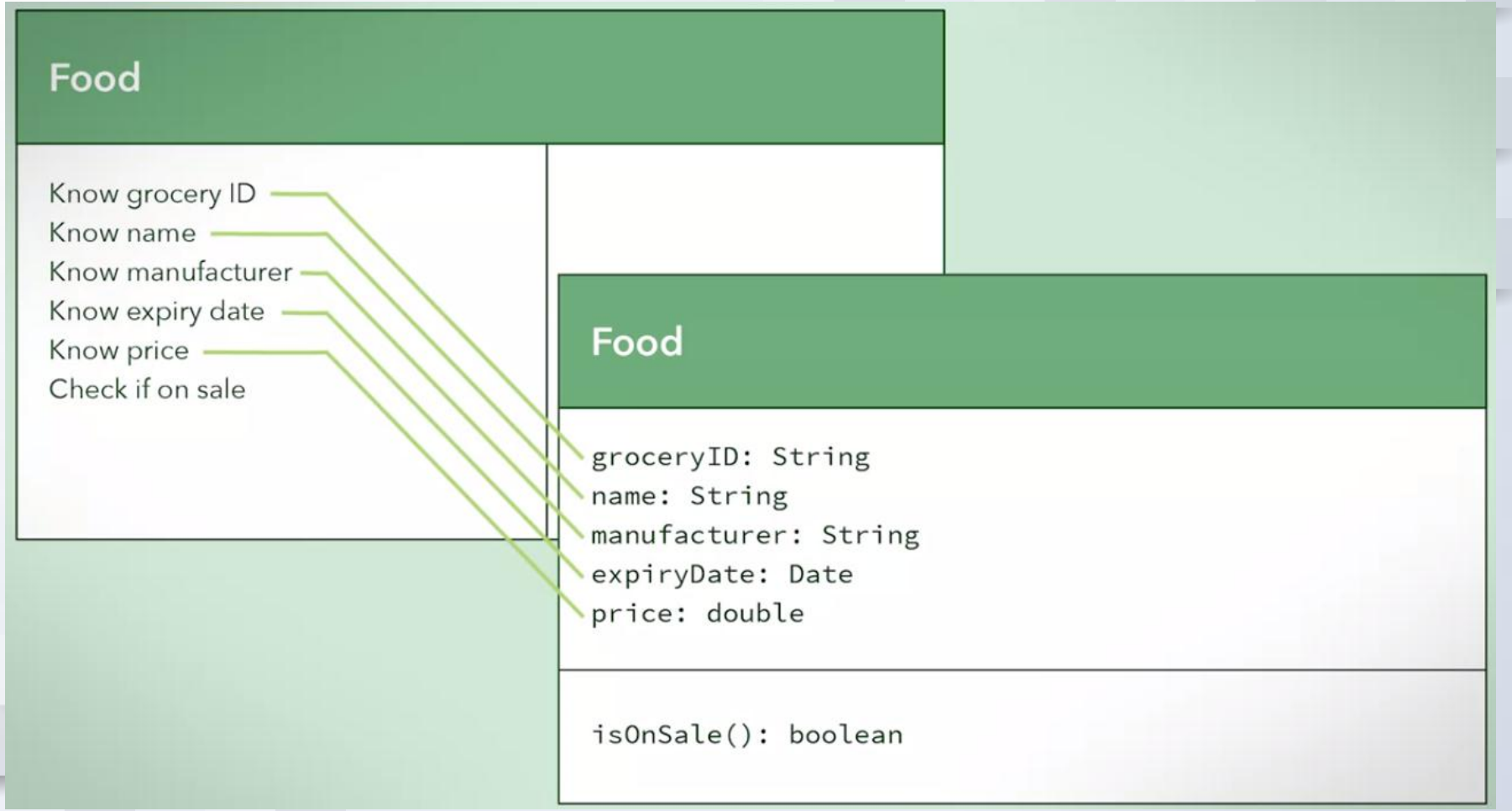
Abstraction...

- Although CRC cards represent **components**, if you remember early on, the goal of design is for the components, when they are refined enough, to become **functions, classes** or **collections** of other components.
- Since we use Java, where an abstraction is formed in a class, we focus on classes.
- So how would the food class look in a Class Diagram?

Class diagram for Food

- This is the Class Diagram representation of the food class.
- Each class in the Class Diagram is represented by a box.
- Each box is divided in three sections much like a CRC card.
- The top part is the **Class Name**.
- This would be the same as the class name in your Java class.
- The middle part is the Property section.
- This would be equivalent to the **member variables in your Java class** and defines the **attributes of the abstraction.**
- And finally, the bottom part is the operations section which is equivalent to the **methods in your Java class** and defines the **behaviors of the abstraction**.




Food
Know grocery ID
Know name
Know manufacturer
Know expiry date
Know price
Check if on sale
Food
groceryID: String
name: String
manufacturer: String
expiryDate: Date
price: double
isOnSale(): boolean

- If we compare the CRC card to our Class Diagram, you might notice how some of the responsibilities on the card turned into properties in the Class Diagram.
- Some, specifically isOnSale, became an operation.
  - You could certainly use CRC cards for abstracting an object such as a grocery food item but there are simply too many ambiguities that prevent a programmer from translating a CRC card to code. One ambiguity is that a CRC card does not show a separation between properties and operations. They are all listed together.

UML Class diagram to Java Code

- Class Diagrams are very close to implementation, making the translation to Java very easy.
- Class name in Class Diagram turns into a class in Java.
- Properties in the Class Diagram turn into member variables. And finally, Operations turn into methods.
- You will probably notice that everything is public. We will assume that for now

Food

groceryID: String
name: String
manufacturer: String
expiryDate: Date
price: double

isOnSale(): boolean

```
public class Food {
    public String groceryID;
    public String name;
    public String manufacturer;
    public Date expiryDate;
    public double price;

    public boolean isOnSale
    () {

    }
}
```

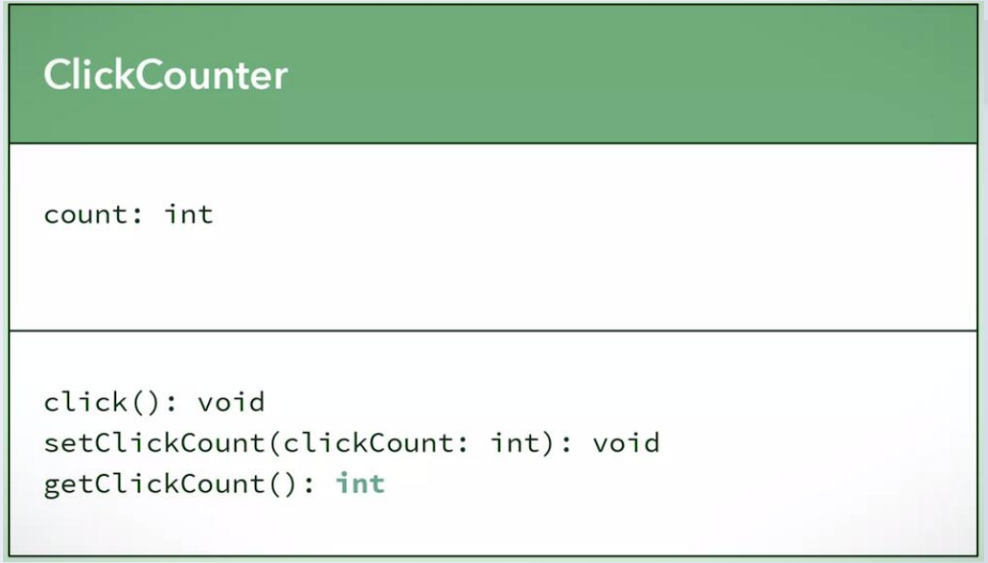
Converting code to Class Diagram

- To convert this code to a Class Diagram, we identify ClickCounter as the class name since that's what the class is named in the code.

```
public class ClickCounter {
  private int count;

  public ClickCounter() {
    count = 0;
  }

  public void click() {
    count++;
  }
  public void setClickCount(int clickCount) {
    count = clickCount;
  }

  public int getClickCount() {
    return count;
  }
}
```

Converting code to Class Diagram

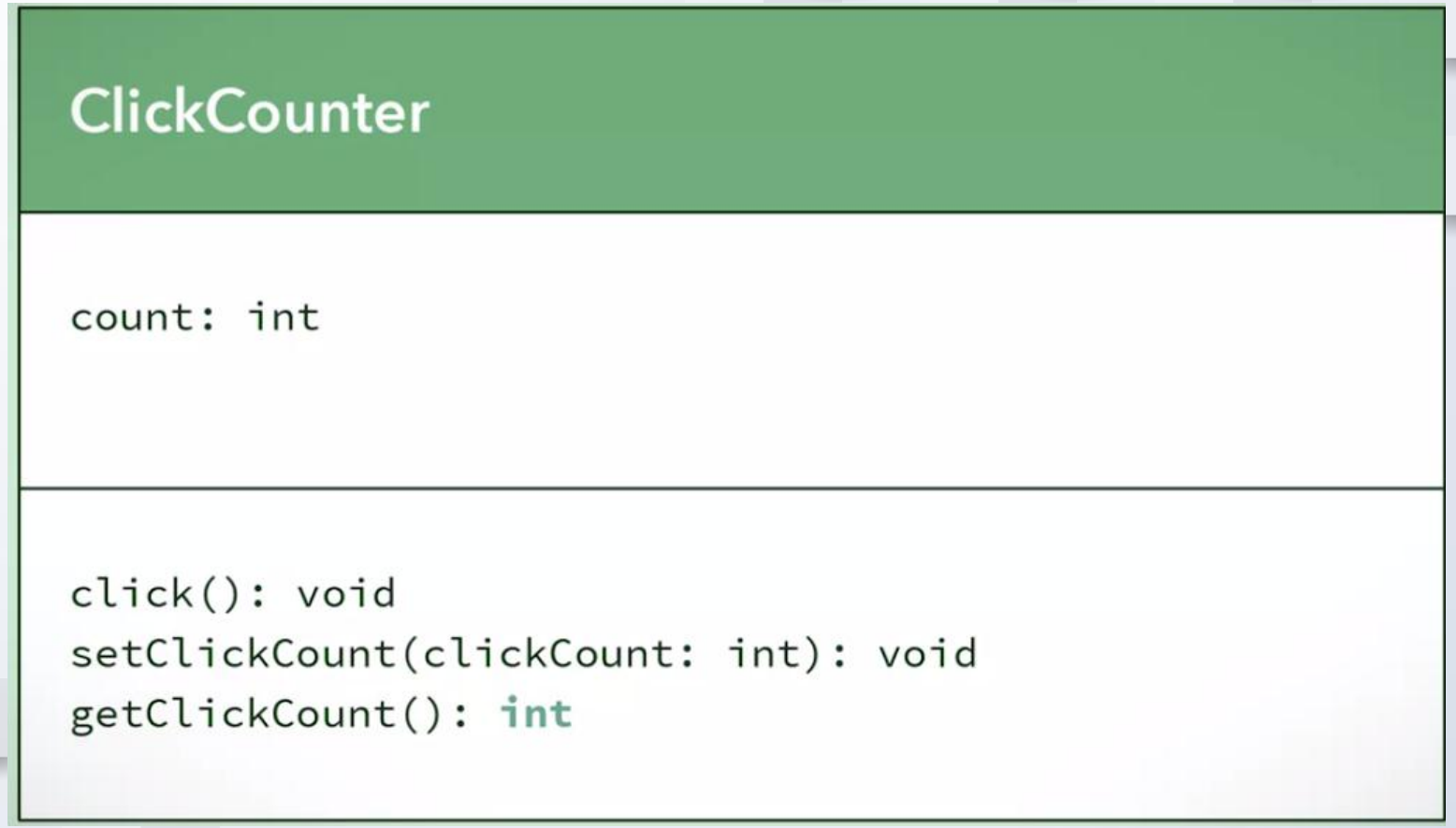
- We, then, set the member variable, count, as a property.
- This property has a type, int.
- Finally, the methods setClickCount and getClickCount become operations.
- setClickCount takes a parameter. Therefore, we will include the parameter list specifying the parameter name and its type. getClickCount has a return value. Therefore, we also have to specify the return type.



Encapsulation in Java and UML

- **Encapsulation** involves three ideas.
- First, you **bundle data,** and **functions** that manipulate the data, into a **self-contained object**.
- Second, you can **expose** certain **data** and **functions** of that object, which can be accessed from other objects.
- Third, you can **restrict access** to certain **data** and **functions** to only within that object.
- So what does that look like in code? And what does the design look like?

- Before we get to the written code, let's take a look at some notation in a UML class diagram that expresses encapsulation.
- If you are creating a system that models a university student using encapsulation, you would have all of the student's relevant data defined in attributes of a student class.
- You would also need specific public methods that access the attributes.
- In this example, our student's relevant data could be their degree program and GPA.


ClickCounter
count: int
click(): void
setClickCount(clickCount: int): void
getClickCount(): int

- This would be the UML class diagram for the student class. The student class has its attributes hidden from public accessibility.
- This is denoted by the minus signs before GPA and degree program. These minus signs indicate that a method or attribute is private. Private attributes can only be accessed from within the class.
- Outside this class, instead of being able to directly manipulate the student's GPA attribute, you must set the GPA through a public method setGPA.
- By only allowing an object's data to be manipulated via a public method, you can control how and when that data is accessed.

Getter Methods

Getter methods are methods that retrieve data, and their names typically begin with get and end with the name of the attribute whose value you will be returning.

Getters often retrieve a private piece of data.

Setter Methods

Setter methods change data, and their names typically begin with set and end with the name of the variable you wish to set.

Setters are used to set a private attribute in a safe way.

- Data integrity is why you have **Getter** and **Setter** Methods.
- In order to change a piece of data, you need to go through the correct channels. Data must be accessed in an approved way.
- Let's take a look at a class that bundles data and has methods that manipulate the data.

- As you can see in this code, whether or not an attribute or method is private or public influences its accessibility.
- Attributes that are private cannot be accessed from anywhere other than from inside the class.
- This hides them from anything outside of the class.
- The only way you can manipulate the hidden data is by writing public functions that allow access to it.

```
public class Student {
  private float gpa;
  private String degreeProgram;

  public float getGPA() {
    return gpa;
  }

  public void setGPA(float newGPA) {
    gpa = newGPA;
  }

  public String getDegreeProgram() {
    return degreeProgram;
  }

  public void setDegreeProgram( String newDegreeProgram ){
    degreeProgram = newDegreeProgram;
  }
}
```

- As you can see in this code, whether or not an attribute or method is private or public influences its accessibility.
- Attributes that are private cannot be accessed from anywhere other than from inside the class. T
- his hides them from anything outside of the class.
- The only way you can manipulate the hidden data is by writing public functions that allow access to it.

Encalpsulation..

- Outside classes do not care how your methods are implemented. They only care if the methods are returning the expected output or doing their expected responsibility.
- This means your Getters and Setters do not purely have to return and change private attribute values. They can do more.
- Let's take a look at our student example.

```
public class Student {
  private float gpa;
  private String degreeProgram;

  public float getGPA() {
    return gpa;
  }

  public void setGPA(float newGPA) {
    gpa = newGPA;
  }

  public String getDegreeProgram() {
    return degreeProgram;
  }

  public void setDegreeProgram( String newDegreeProgram ){
    degreeProgram = newDegreeProgram;
  }
}
```

- You might have a situation in which there are restrictions on changing the degree program.
- You can change your code to more accurately display this restriction when it comes to changing the degree program.
- Let's say that you need a GPA greater than two point seven in order to be able to change the degree program, this restriction might look like this.

```
public class Student {
  private float gpa;
  private String degreeProgram;

  public float getGPA() {
    return gpa;
  }

  public void setGPA(float newGPA) {
    gpa = newGPA;
  }

  public String getDegreeProgram() {
    return degreeProgram;
  }

  public void setDegreeProgram( String newDegreeProgram ){
    if (gpa > 2.7) {
      degreeProgram = newDegreeProgram;
    }
  }
}
```

- The outside observer does not need to know how your public methods are implemented.
- Just that they perform as expected.
- That means you can add additional code to your Getters and Setters if you need to.
- Overall encapsulation is meant to protect your class and its objects. It also allows for an interface of approved methods for other classes to safely use the class.
- It allows you to hide implementation details from other classes.

Decomposition in Java and UML

Decomposition

Taking a **whole** thing and dividing it up into different **parts**. Or, on the flip side, taking a bunch of separate parts with different functionalities and combining them together to form a whole.

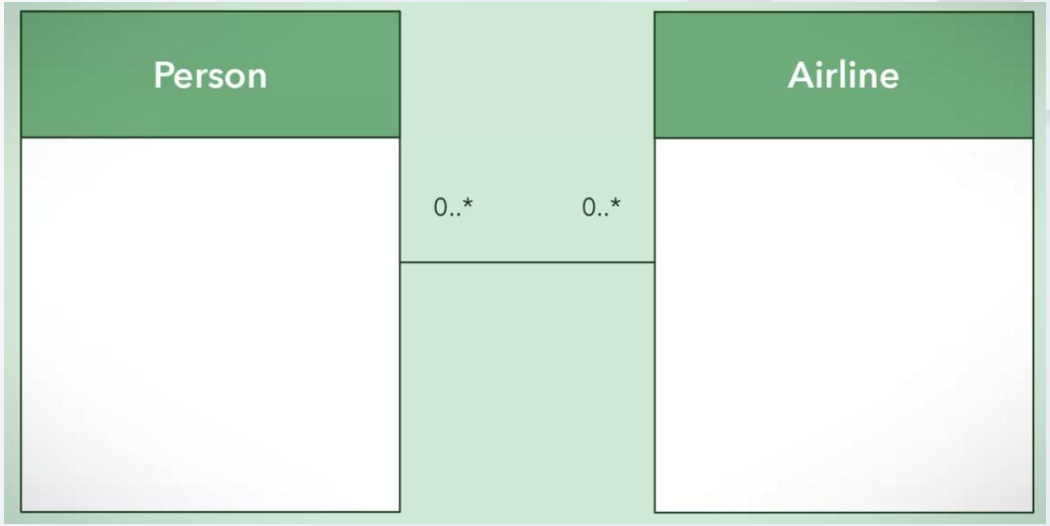
There are three types of relationships found in decomposition:

- Association
- Aggregation
- Composition

Association

Association is “some” relationship. This means that there is a loose relationship between two objects. These objects may interact with each other for some time.

- For example, an object of a class may use **services/methods** provided by object of another class.
- This is like the relationship between person and airline.
- A person does not generally own an airline, but can interact with one.
- An airline can also interact with many person objects.
- There are some persons and some airlines, **neither is dependent on the other**.



- The straight line between two UML objects denotes that the relationship between them is an association.
- You can see that there is a 0..* found on both sides of the relationship.
- This means a person object is associated with zero or more airline objects and an airline object is associated with zero or more person objects.

Association

- The association between food and wine or students and sports.
- Each of these relationships is between completely separate entities.
- If one object is destroyed, the other can continue to exist, unlike human and organ.
- There can be any number of each item in the relationship.
- One object does not belong to another.

Association Java Code

```
public class Student{
    public void play(Sport sport){
        …
    }
}
```

- In this code excerpt, the student is passed a sport object to play.
- The student does not possess the sport beyond playing it.
- The relationship is between two completely separate objects.
- A student can play any number of sports. And any number of students can play a sport.
- Now it's your turn to come up with some code displaying an association.

```
public class Wine{
	public void pair(Food food){
		execute.pair(food);
	}
}
```

- As with our student example, the wine can exist independent of the food. It does not need food to exist, nor does it always have food.
- The two objects interact with each other without belonging to one another.
- Overall, association is a loose partnership between two objects that exist completely independently.
- They have numbers that are not tied to each other.

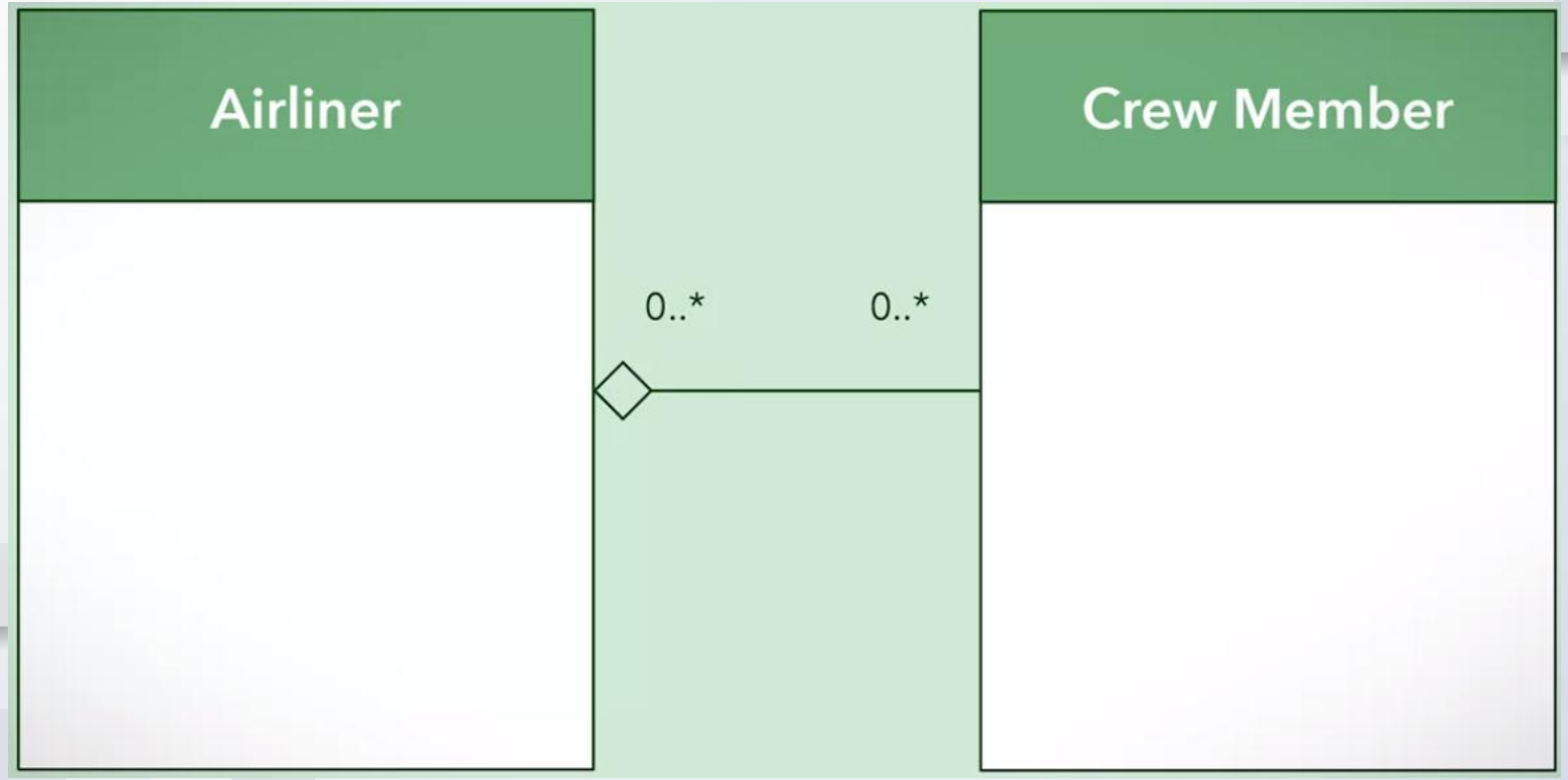
Aggregation

Aggregation is a "has-a" relationship where a whole has parts that belong to it. There may be sharing of parts among the wholes in this relationship.

The "has-a" relationship from a whole to the parts is considered weak. What this means is although parts can belong to the wholes, they can also exist independently.

- This is like the relationship between an airliner and its crew. An important part of the airliner is its crew, without the crew, an airliner would not be able to fly.
- However, the airliner does not cease to exist if there is no crew on board.
- Same goes for the crew, they are part of the operation of the airliner but the crew does not cease to exist or become destroyed if they are not on board their airliner.
- These entities have a relationship, but can exist outside of it.

UML Class diagram
Aggregation



UML Class diagram Aggregation

- This UML class diagram describes the relationship I explained above between airliner and crew.
- It says that for an airliner object, it has zero or more crew members.
- Also, a crew member object can be had by zero or more airline objects.
- The empty diamond denotes which object is considered the whole and not the part in the relationship.
- This empty diamond is the symbol for aggregation.

Aggregation

- The aggregation we can see is between course section and students, pet stores and pets, and bookshelf and books.
- Each of these are a has-a relationship. A course has students, students have courses, and so on.
- But the relationship is weak. If one of the objects in the relationship is destroyed, it still makes sense that the other can continue to exist.

Aggregation Java Code

```
public class Airliner {
     private ArrayList<CrewMember> crew;

     public Airliner(){
          crew = new ArrayList<CrewMember>();
     }

     public void add( CrewMember crewMember ){ … }
}
```

- n the airliner class, there is a list of crew members.
- The list of crew members is initialized to be empty and a public method allows new crew members to be added.
- The airliner has a crew. This means that an airliner can have zero or more crew members.

So..

- Aggregation is a weak has-a relationship between classes.
- One object has the other, but the objects are not heavily linked.
- They can both exist without the other.

One of the major decomposition relationships is composition.

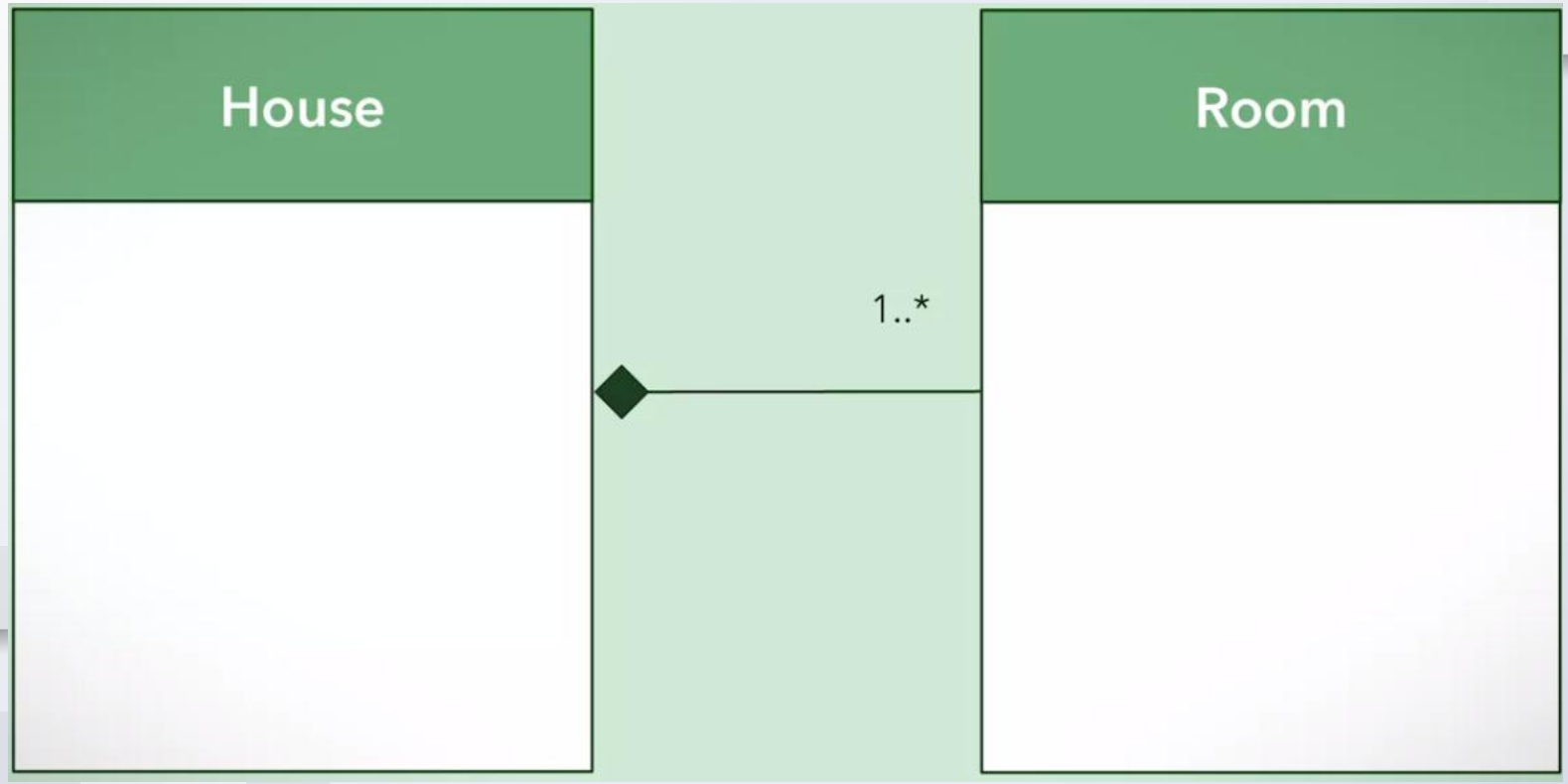
Composition

Composition is an exclusive containment of parts, otherwise known as a **strong** has-a relationship. What this means is that the whole cannot exist without its parts. If it loses any of its parts the whole ceases to exist. If the whole is destroyed then all of its parts are destroyed too. Usually you can only access the parts through its whole. Contained parts are exclusive to the whole.

Composition

- Compare this to the relationship between a house and a room.
- A house is made up of multiple rooms.
- However, if you were to remove the house, its rooms would cease to exist. You cannot have a room without its house.

UML Class Diagram
Composition



UML Class Diagram Composition

- This UML class diagram describes the relationship between a house and a room, that a house object has one or more room objects.
- The filled in diamond next to the house means that the house is the whole in the relationship.
- If the diamond is filled in, it means that has-a relationship is strong. The two related objects cannot exist without each other.
- The filled diamond denotes the relationship is composition.

Choose the object pair that best describe a composition.

- (●) Human - Brain
- ( ) Dog - Collar
- ( ) Pencil Case - Pencil
- ( ) Nail - Nail Polish

- The composition is between the human and brain pair.
- This relationship is between completely dependent classes.
- If one object is destroyed, then the other is too.

```
public class Human{
    private Brain brain;

    public Human(){
        brain = new Brain();
    }
}
```

Composition Java Code

- In this example, the brain is created at the same time that the human object is.
- The brain does not need to be instantiated anywhere else, nor does it need to be passed into the human object on creation.
- The brain is automatically created with the human. The two parts, human and brain, are tightly dependent with one not being able to exist without the other.

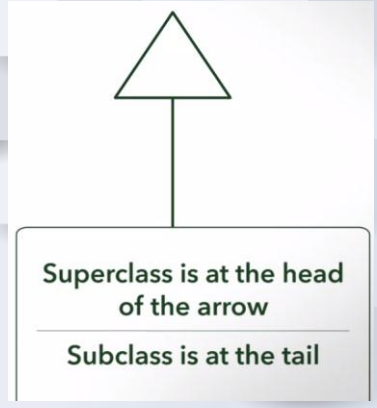
Composition

- This example is the same as our human and brain example.
- The employee cannot exist without a salary and the salary cannot exist without an employee.
- On instantiating an employee, the salary part is made.
- The salary must always exist as long as the employee does from that point on.
- Composition is the most dependent of the decomposition relationships.
- It forms a relationship that only exists as long as each object exists.

Decomposition Summary

- Decomposition is simply about whole objects containing part objects.
- Depending on your design, you can relate wholes to parts in different increasingly tighter ways.
- You can use association, a very loose interaction between two completely independent objects.
- An aggregation, one whole has a part, but both can live independently.
- And finally, in composition, the whole cannot exist without its parts and vice versa.
- All three relationships are useful and versatile for your software designs.

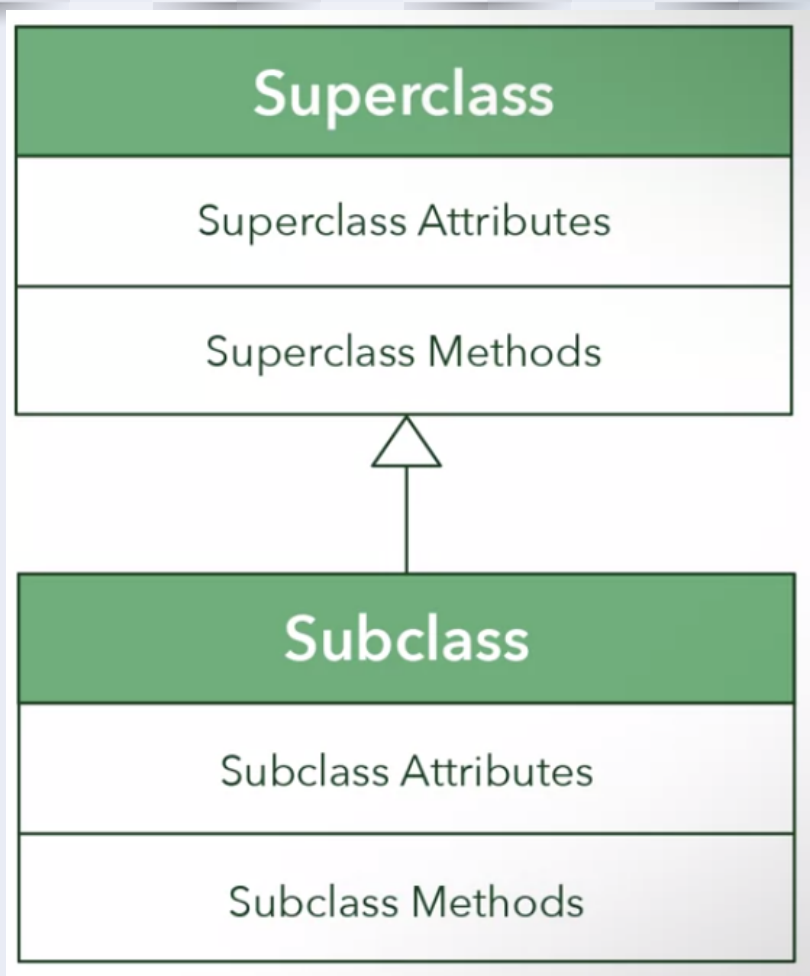
Generalization in Java and UML



- Showing inheritance is very simple in a UML class diagram.
- You simply connect two classes with a solid lined arrow.
- This indicates, that two classes are connected by inheritance.
- The superclass is at the head of the arrow, and the subclass is at the tail.
- The standard way to draw inheritance into your UML diagrams, is to have the arrow pointing upward.
- This means that the superclasses are always toward the top, and the subclasses are always toward the bottom.

UML Class Diagram Generalization

- You do not need to put any of the inherited superclasses attributes and behaviors into the subclass.
- The arrow is used to communicate inheritance, which implies that the subclass will have the superclasses attributes and methods.
- The superclasses are the generalized classes, and the subclasses are the specialized classes.

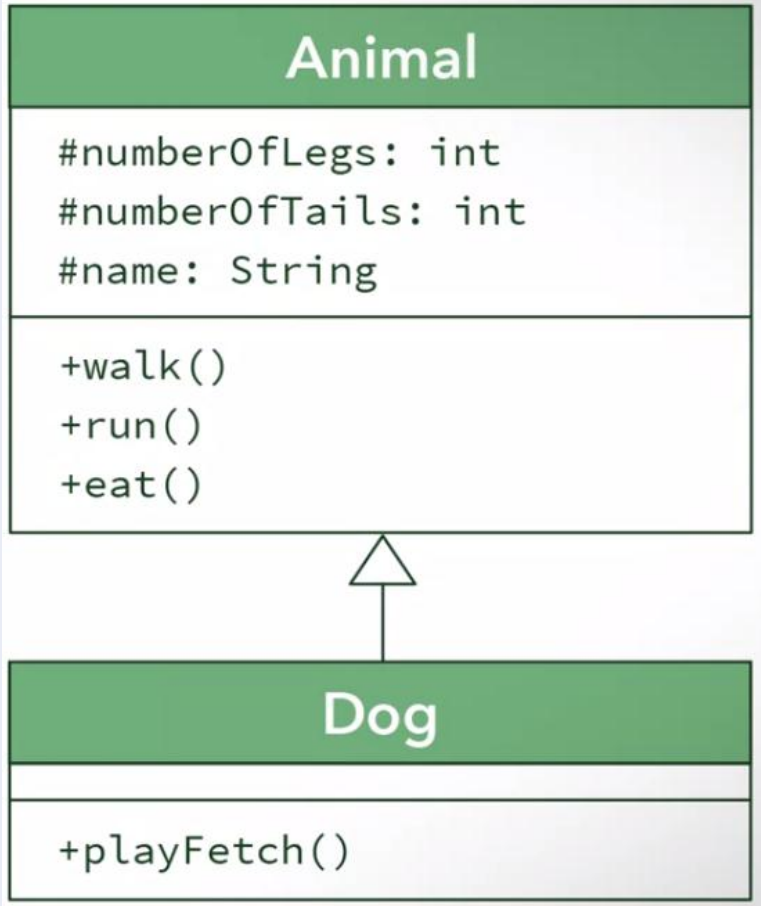


Generalization

- Suppose you have a dog class and an Animal class.
- Are you able to see which class is a superclass and which one is the subclass?
- Is the dog class the superclass or the subclass?
- First, we will model the dog and animal classes in a UML diagram to show the relationship between them.
- This will show how the two classes are related to each other, how the superclass is generalized, and how the subclass is specialized.

Class Diagram Generalization

- A UML class diagram describes a dog class as a subclass, and the Animal class as the superclass.
- This means that the dog class will inherit from the Animal class.
- The hash symbol is used to communicate that the animals attributes are protected.



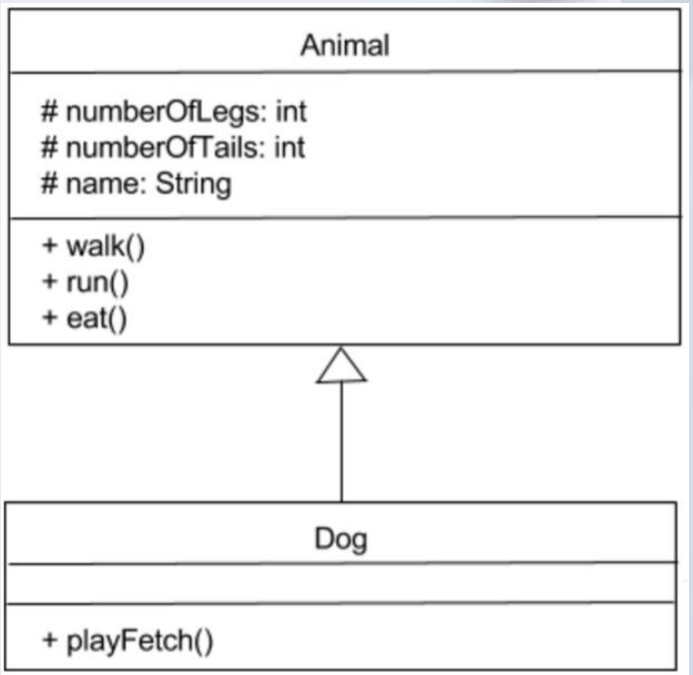
In Java, a protected attribute or method can only be accessed by:

- The encapsulating class itself
- All subclasses
- All classes within the same package

- In Java, a package is simply a means in which the classes can be organized into a namespace that represents those classes.

Quiz!

- In our UML class diagram, which attributes and behaviors of the Animal class are inherited by the Dog class? (Note: You can assume that the two classes are in the same package.)



- ○ None of the attributes and behaviors are inherited.
- ● All attributes and behaviors of Animal are inherited.
- ○ playFetch()
- ○ numberOfLegs, numberOfTails, walk(), run(), eat()

Generalization
Java Code

```
public abstract class Animal {
     protected int numberOfLegs;
     protected int numberOfTails;
     protected String name;

     public Animal(String petName, int legs, int tails) {
          this.name = petName;
          this.numberOfLegs = legs;
          this.numberOfTails = tails;
     }

     public void walk() { … }
     public void run() { … }
     public void eat() { … }
}
```

Generalization
Java Code

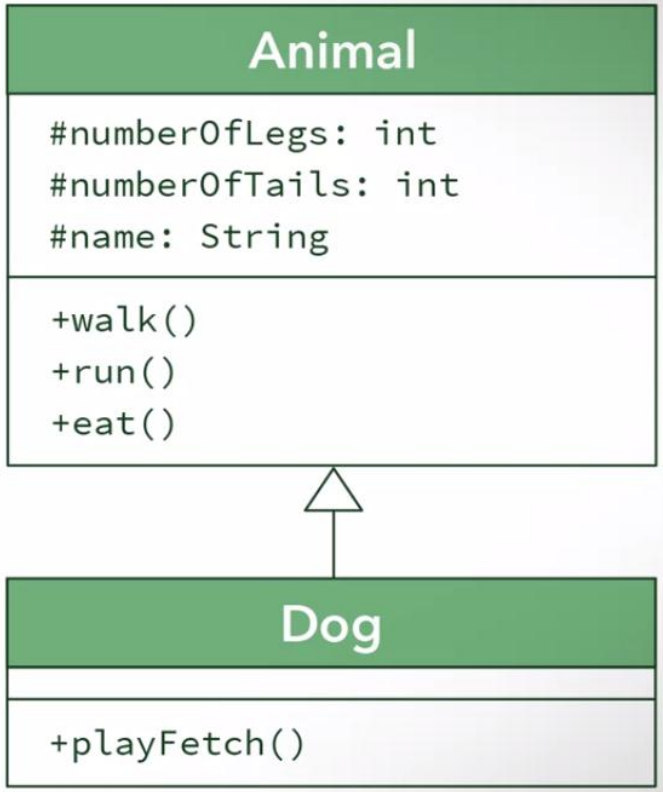
```
public abstract class Animal {
     protected int numberOfLegs;
     protected int numberOfTails;
     protected String name;

     public Animal(String petName, int legs, int tails) {
          this.name = petName;
          this.numberOfLegs = legs;
          this.numberOfTails = tails;
     }

     public void walk() { … }
     public void run() { … }
     public void eat() { … }
}
```

~~Instantiated~~

- Since an animal is a generalization of specific species, we do not want to be able to create an animal object on its own.
- We use the keyword abstract to declare that this class cannot be instantiated.
- That means that we cannot create an animal object.

- The Animal class will be the superclass for our dogs subclass, any class that inherits from the Animal class will have its attributes and behaviors.
- This means that if we were to introduce a cat subclass into our system that inherited from the Animal class, the cat and dog subclasses would both have the same attributes and behaviors as the animal superclass.

```
public class Dog extends Animal {
     public Dog(string name, int
     legs, int tails) {
          super(name, legs, tails);
     }

     public void playFetch() { … }
}
```



- We declare inheritance in Java using the keyword "extends".

- You instantiate objects from a class by using constructors.
- With inheritance, if you want an instance of a subclass, you need to give the superclass a chance to prepare the attributes for the object appropriately.
- With inheritance, if you want an instance of a subclass, you need to give the superclass a chance to prepare the attributes for the object appropriately.

- Classes can have implicit constructors or explicit constructors. In this implementation of the Animal class, we have an implicit constructor, since we have not written our own constructor.

```
public abstract class Animal {
    protected int numberOfLegs;

    public void walk() { … }
}
```

- All attributes are assigned zero or null, when using the default constructor.

- The Animal class in this implementation, has an explicit constructor that will let us instantiate an animal with however many legs we want.

```
public abstract class Animal {
    protected int numberOfLegs;

    public Animal (int legs) {
        this.numberOfLegs = legs;
    }
}
```

Explicit constructors are used so that we can assign values to attributes during instantiation

- This is because explicit constructors of the superclass must be referenced by the subclass.
- Otherwise the superclass attributes would not be appropriately initialized.

```
public abstract class Animal {
    protected int numberOfLegs;

    public Animal (int legs) {
        this.numberOfLegs = legs;
    }
}

public class Dog extends Animal {
    public Dog(int legs) {
        super(legs);
    }
```

A subclass's constructor must call its superclass's constructor if the superclass has an explicit constructor

- In order to access the superclass's attributes, methods and constructors, the subclass uses the keyword called Super.

```
public abstract class Animal {
    protected int numberOfLegs;

    public Animal (int legs) {
        this.numberOfLegs = legs;
    }
}

public class Dog extends Animal {
    public Dog(int legs) {
        super(legs);
    }
}
```

Override

- Subclasses can override the methods of its superclass, meaning that a subclass can provide its own implementation for an inherited superclass's method.
- The dog class has overwritten the animal class's walk method. If we were to ask the dog to walk, it would tell us that it would rather lay on the couch instead of performing the behavior implemented in the Animal class

Override

```
public abstract class Animal {
     protected int numberOfLegs;

     public void walk() {
          System.out.println(“Animal is walking”);
     }
}

public class Dog extends Animal {
     public void walk() {
          System.out.println(“I’d rather lay on the couch”);
     }
}
```

Quiz!

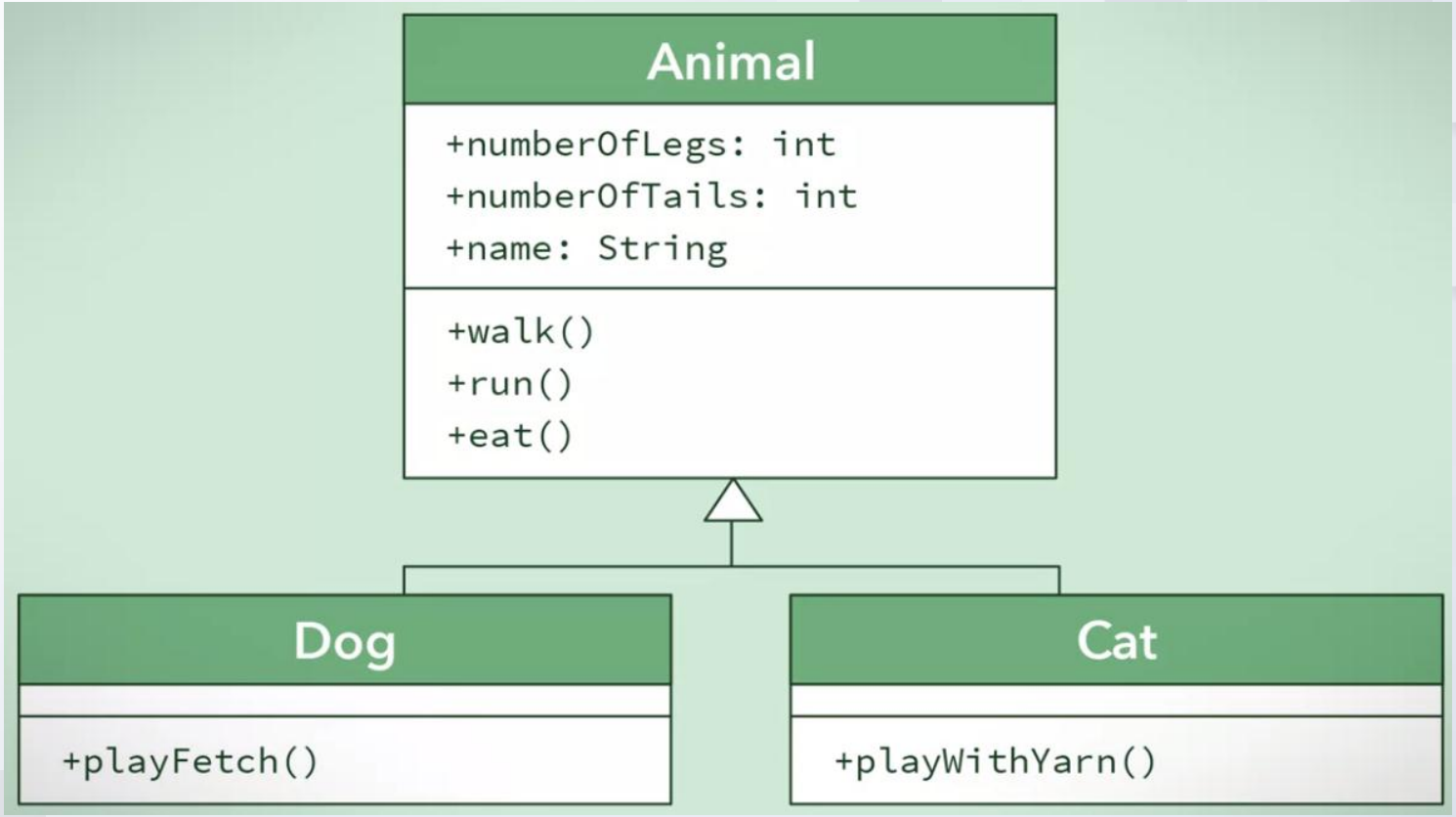
You are a developer in charge of creating different methods of travel in a video game. The team has decided to give the player various options for transportation, which include: riding a horse, driving a car, or flying a plane. You have decided that you will generalize these modes of transportation into an abstract class called Transport.

Which attributes and behaviors would you include in your general class?

- [ ] public void verticalMovement() { … }
- [ ] public int fuel;
- [x] public void moveForward() { … }
- [x] public int speed;

Inheritance Types

- The inheritance that we have been looking at is called implementation inheritance.
- For Java, only single implementation inheritance is allowed.
- Well a superclass can have multiple subclasses.
- A subclass can only inherit from one superclass.
- For example, the dog and cat subclasses can each only have implementation inheritance with one superclass, which is animal.



- The animal superclass however, can have any number of subclasses, two in this example.

- To implement this in code, we simply have the cat and dog classes extend the Animal class.

```
public class Cat extends Animal {
     public Cat(string name, int legs, int tails) {
          super(name, legs, tails);
     }

     public void playWithYarn() { … }
}

public class Dog extends Animal {
     public Dog(string name, int legs, int tails) {
          super(name, legs, tails);
     }

     public void playFetch() { … }
}
```

- Now we have a cat and a dog that behave like an animal without having to explicitly write code for them.
- They also have their own behaviors. Doug would know how to play fetch, but would not know how to play with yarn like Mittens would.
- In this way, we can create specialized classes, like the dog and cat subclasses, with customized or special behaviors.
- Note, that a subclass itself can be a superclass to another class. Inheritance can trickle down through as many classes as you want.
- Inheritance will let you generalize related classes into a single superclass and still allow the subclasses to retain the same set of attributes and behaviors.
- This will help remove redundancy in your code, and make it easier to implement changes.

Generalization with Interfaces in Java and UML

Sub-typing

- A class denotes a type for its objects.
- The type signifies what these objects can do through public methods.
- For example, instances of a dog class are dog typed objects, and these objects do dog things.
- In modeling a problem, we may want to express subtyping relationships between two types.
- For example, we can have dog type as a subtype of animal type.
- This means a dog object is not only dog typed, it is also animal typed.
- So a dog object behaves not only like a dog, it should also behave like an animal, in effect, a dog is an animal.

Sub-typing

- In JAVA, class inheritance with the extends keyword is often used for subtyping.
- If a dog subclass extends an animal superclass, a dog object behaves not only like a dog, it will also behave by default like an animal through the inherited methods and attributes of an animal.
- In effect, a dog is an animal.
- Here, the dog class inherits the implementation details of animal class.

- A JAVA interface also denotes a type.
- Unlike a class, however, an interface only declares method signatures, and no constructors, attributes, or method bodies.
- It specifies the expected behaviors in the method signatures, but does not provide any implementation details.

- In JAVA, an interface is also used for subtyping.
- If a dog class implements an I animal interface, then a dog object behaves not only like a dog, but it is also expected to behave like an animal by providing all the method bodies for the method signatures listed in the interface.

- Just like with inheritance, the dog is an animal. However, the difference is that the dog class needs to provide the implementation details for what it means to be an animal.
- So, an interface is like a contract to be fulfilled by implementing classes.

- In both inheritance and interfaces, you achieve consistency between the dog type and the animal type so that a dog object is usable anywhere in your program when you are dealing with an animal type.
- Unlike inheritance, interfaces are not a generalization of a set of classes. It is important to understand that interfaces are not classes. They are used to describe behaviors.
- All that an interface contains are method signatures.

```
public interface IAnimal {
    public void move();
    public void speak();
    public void eat();
}
```

- In JAVA, we use the key word interface to indicate that we are creating one.
- Standard JAVA naming convention places the letter I before an actual name to indicate an interface.
- This interface describes three different behaviors of an animal, which are moving, speaking, and eating.
- Notice how we never implement or describe how these behaviors are performed.
- We only show that an animal has these behaviors.

- Another thing you might have noticed is that the interface does not encapsulate any of the attributes of an animal.
- This is because attributes are not behaviors.
- Now that we have an interface, how do we use it?
- We need to declare that we are going to fulfill the contract as described in the interface.
- The keyword in JAVA for this action is **implements**.

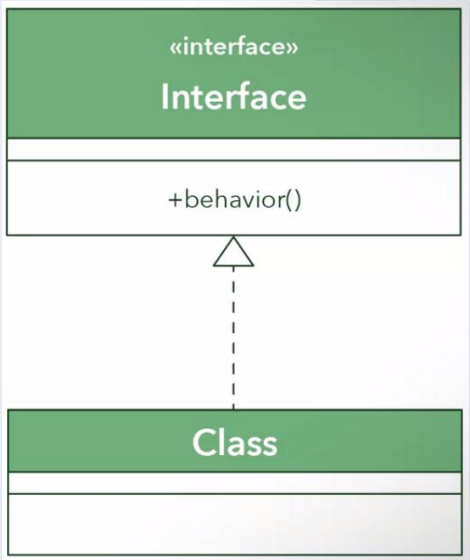
- Our dog class has declared that it will implement or describe the behaviors that are in the interface.
- When you do this, you must have all the method signatures explicitly declared and implemented in the class.
- This means that we must the move, speak, and eat methods in this class.

```
public class Dog implements IAnimal {
    /* Attributes of a dog can go here */

    public void move() { … }

    public void speak() { … }

    public void eat() { … }
}
```

You must have all the method signatures explicitly declared and implemented in the class

UML Diagram for Interface

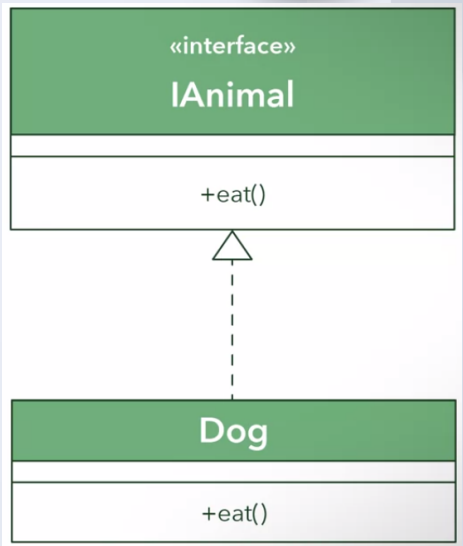
- Interfaces are drawn in a similar way that classes are drawn in UMLs. Interfaces are explicitly noted in UML class diagrams using guillemets, or French quotes, to surround the words interface.
- The interaction between an interface and a class that is implementing the interface is indicated using a dotted arrow.
- The class touches the tail end of the arrow and the interface touches the head of the arrow.



- We combine these notations together with a class to show that a class implements an interface.
- This indicates that the class implements the interface.
- The standard way to draw interfaces on your UML class diagrams is to have the arrow pointing upward.
- This means that the interface is always toward the top, and the classes that implement them are always toward the bottom.

UML Diagram for Interface

- This UML class diagram tells us that the Dog class will determine how the behavior that is described in the interface IAnimal will be implemented by repeating the method signature.
- There are several advantages for interfaces. Knowing and understanding what these advantages are will help you to determine if you should use interfaces or use inheritance when you are designing your systems.



- Like abstract classes, which are classes that cannot be instantiated, interfaces are a means in which you can implement polymorphism.
- In object oriented languages, polymorphism is when two classes have the same description of a behavior, but the implementations of the behavior may be different.
- This can be seen when we compare a cat and a dog. How would you describe how each of these animals speak? Well, to simply put it, a cat meows and a dog barks.
- The description of the behavior is the same, both animals can speak. But the actual behavior implementation itself is different.
- This is known as **polymorphism**. It is simple to achieve in JAVA using an interface.

Polymorphism

- We create our interface the same way as we did before.
- The Cat and Dog class both implement the IAnimal interface, but they each have their own versions of the speak behavior. When we ask Doug the Dog to speak, he knows how to bark, but will not know how to meow like Mittens the Cat.

```
public interface IAnimal {
    public void move();
    public void speak();
    public void eat();
}

public class Dog implements IAnimal {
    public void speak() {
        System.out.println("Bark!");
    }
}

public class Cat implements IAnimal {
    public void speak() {
        System.out.println("Meow!");
    }
}
```

- Just like with class inheritance, interfaces can inherit from other interfaces and just like with class inheritance, interface inheritance should not be abused.
- This means that you should not be extending interfaces if you are simply trying to create a larger interface.
- Interface A should only inherit from interface B if the behaviors in interface A can fully be used as a substitution for interface B.

Example

```
public interface IVehicleMovement {
    public void moveOnX();
    public void moveOnY();
}
```

- Lets simplify the movement of a vehicle by restricting it s movement so that it can only travel along either the x-axis or y-axis.
- This interface can be used to describe the behaviors of vehicles on land or on water.
- But what if we need to implement the movement of a plane or a submarine that can also move in the zed-axis? We do not want to add an extra behavior to the interface, because on-land and on-water vehicles do not move along the zed-axis. So what do we do?

Example

```
public interface IVehicleMovement3D
extends IVehicleMovement {
     public void moveOnZ();
}
```

- We can create a second interface that will inherit from our first one.
- Now, we can use the IVehicleMovement3D interface for all vehicles that have three-dimensional movement without having to add
- the Zed-axis movement to the interface used by the on-land and on-water vehicles.

Multi-interfaces implementations

- In JAVA, a class can implement as many interfaces as we want.
- This is because of the nature of interfaces.
- Since they are only contracts and do not enforce a specific way to complete these contracts, overlapping method signatures are not a problem.
- A single implementation for multiple interfaces with overlapping contracts is acceptable.

```
public interface IPublicSpeaking {
     public void givePresentation();
     public void speak();
}

public interface IPrivateConversation {
     public void lowerVoiceVolume();
     public void speak();
}

public class Person implements IPublicSpeaking, IPrivateConversation {
     public void speak() {
          System.out.println(“This is fine”);
     }
}
```

- A single implementation for multiple interfaces with overlapping contracts is acceptable.
- There is no ambiguity here because the Person class only has one definition of a speak method, and it is the same implementation for both interfaces.
- This is JAVA's approach to avoid the issue that is introduced with multiple inheritance.

- Interfaces are powerful tool to allow you describe a set of behaviors.
- Classes can implement one or more interface at a time which allows them to have multiple types. Interfaces enable you to describe behaviors without the need to implement them, which allows you to reuse these abstractions. Just like with other constructs in object oriented modeling and programming, interfaces will help you to create programs with reusable and flexible code.
- Although they are a useful technique, remember that you should not be generalizing all behavior contracts into interfaces.
- They are meant to fulfill a specific need, which is to provide a way for related classes to work consistently.