Unit 13: Object-Oriented Programming (OOP)

Anthony Estey

CSC 115: Fundamentals of Programming II

University of Victoria

Unit 13 Overview

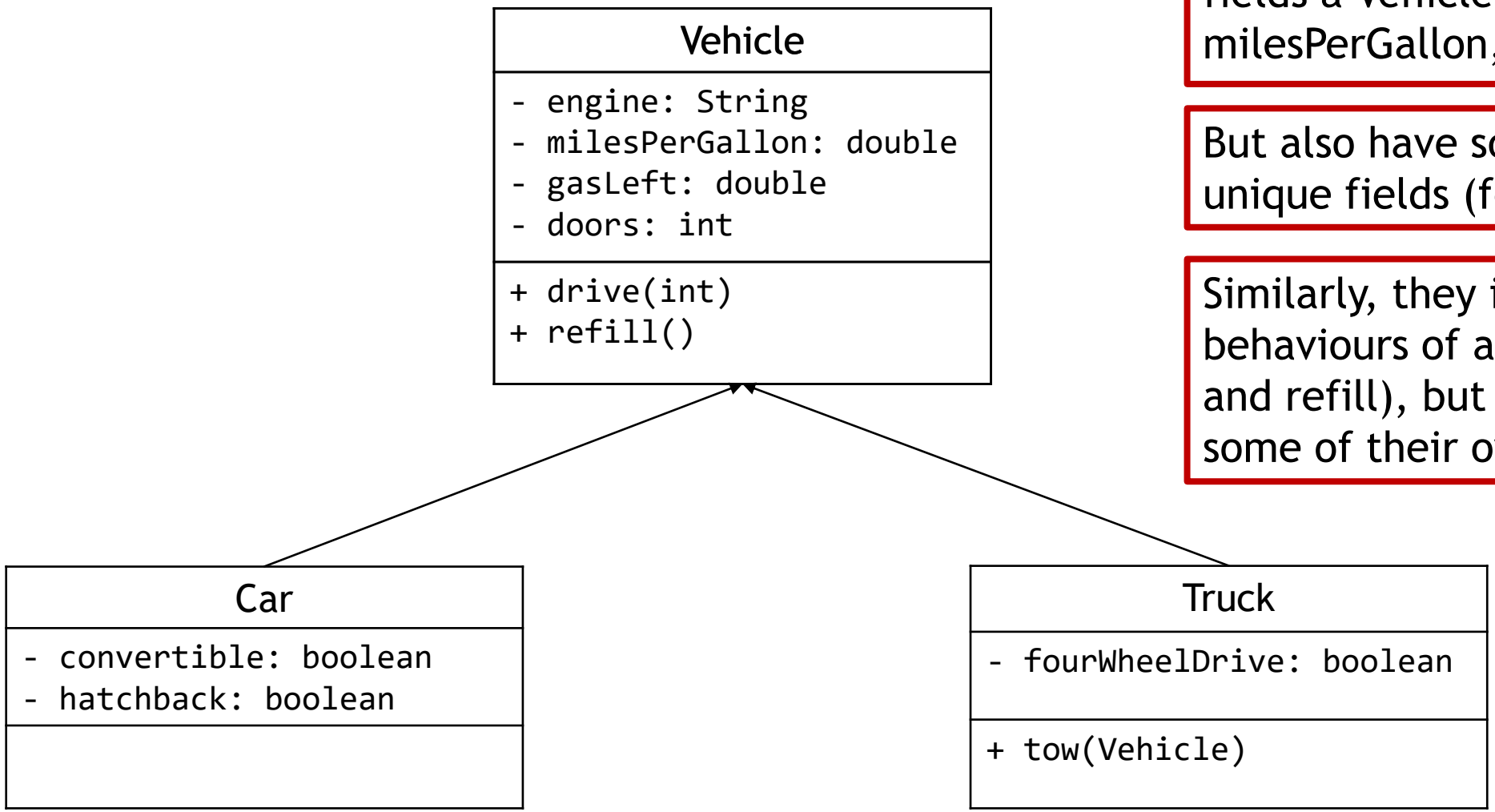
- Related Reading:
  - Textbook: Pages 95-102, 218-224, 456-466
- Learning Objectives: (You should be able to…)
  - describe the concept of encapsulation, and how objects enable encapsulation in Java
  - describe the concept of inheritance and how inheritance relations work in Java
  - read and write Java code that uses inheritance, and identify which fields and methods are shared across classes associated with the inheritance
  - describe how the extends and super keywords are used in Java
  - describe the concept of polymorphism, as well as when casting in necessary

Inheritance - Motivation

- One of the reasons we write methods, or functions, is to reduce redundancy and code repetition
- Sometimes when we create classes there is a lot of repetition too

- For example:
  - Within a school system there may be Instructors, Students, and Staff members each with their own class
  - But all three of these classes share some things in common
  - This is one of the problems inheritance solves

Inheritance Example



Trucks and Cars inherit all the fields a Vehicle has (engine, milesPerGallon, etc)

But also have some of their own unique fields (fourWheelDrive)

Similarly, they inherit some behaviours of a Vehicle (drive and refill), but might also have some of their own (tow)

Java Inheritance Terminology

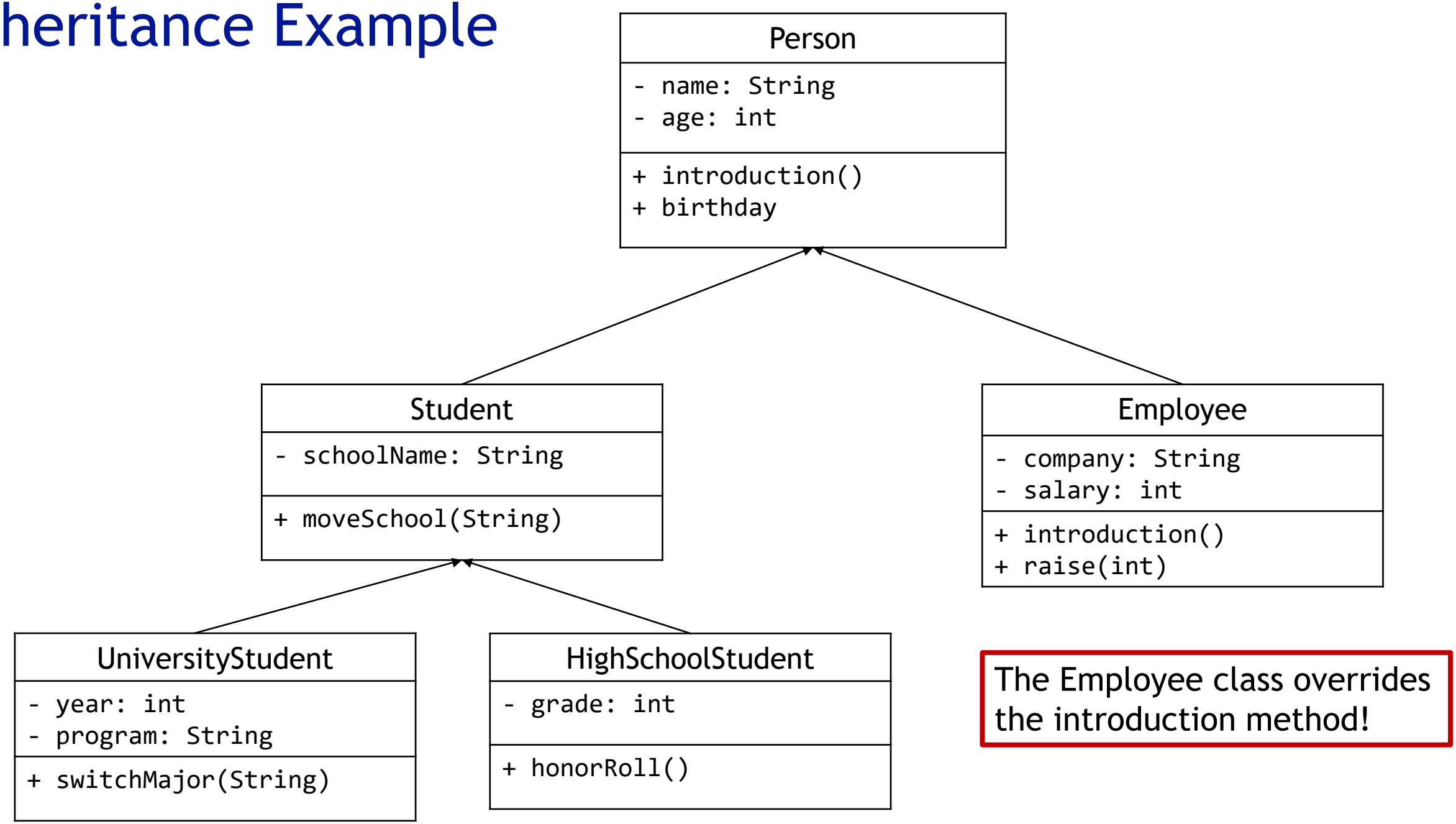
- **Inheritance**: refers to the ability for one class to inherit from another

- When a class inherits from another, we say it ***extends*** the other class
  - a **subclass** extends (inherits from) a **superclass**
  - a subclass is often sometimes called a *specialization* of a superclass

Subclasses

- A subclass can:
  - Add new fields in addition to those it inherits
    - a subclass inherits private fields, but cannot access them directly
  - Can call/invoke methods found in the superclass
  - Override an inherited method of its superclass
    - this occurs when a method in the subclass has the same method name and signature as a method in it superclass
    - Sometimes subclasses *refine* a method from the superclass

Inheritance Example



The Employee class overrides the introduction method!

Inheritance Example

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void introduction() {
        System.out.println("Hello, I'm "+name+". I'm "+age+" years old");
    }
}
```

```java
Person p1 = new Person("Ali", 22);
p1.introduction();
```

Output: Hello, I'm Ali. I'm 22 years old

```java
public class Employee extends Person {
    private String company;
    private int salary;

    public Employee(String name, int age, String company, int salary) {
        super(name, age);
        this.company = company;
        this.salary = salary;
    }

    public void introduction() {
        super.introduction();
        System.out.println("I work at "+company);
    }
}
```

```java
Employee e1 = new Employee("Sam", 28, "Microsoft", 950000);
e1.introduction();
```

Output: Hello, I'm Sam. I'm 28 years old
I work at Microsoft

Object-Oriented Programming

- We have now created our own objects, and more recently, used inheritance to extend our objects into subclasses
  - these are two important parts of object-oriented programming (OOP)
  - but there is more to OOP than simply working with objects

- The four principles of object-oriented programming:

1. **Encapsulation**: objects combine data and operations into a single unit
2. **Abstraction**: unnecessary details are hidden
3. **Inheritance**: classes can inherit properties from other classes
4. **Polymorphism**: objects determine appropriate operations at runtime

1. Encapsulation

- When writing software to solve problems, we need to represent many different kinds of information
  - Sometimes primitive types (ints, chars, booleans, etc) are enough
  - Sometimes the information we are working with contains two or more values naturally belong together

- Some examples:
  - the x and y position of a **point** in a graph
  - the *title*, *artist*, and *duration* of a **song**
  - the *name*, *ID*, *gpa*, and *program* for a **student**

1. Encapsulation

- In Java, we have created classes to solve certain problems
  - for example, all of the data we need to represent student information can be written as fields within a Student class
  - the (non-static) methods in the class allow us to operate on that data
  - these methods make up the behaviours of the class

- One part of **encapsulation** is this bundling of data and code that operates on that data into a single unit
  - a class!

- The other part is the fact that we can restrict access to that data

1. Encapsulation

- In Java, we use **access modifiers** to control whether other classes can use a particular field or call a particular method

- The **access modifiers** we have seen so far:
  - **private**: can only be accessed from within the same class
  - **protected**: can be accessed from within the same class, package, or folder
  - **public**: can be accessed from everywhere

- Encapsulation is used to hide certain values within a class
- Values are accessed or modified through publically accessible methods

1. Encapsulation

- Encapsulation is used to hide certain values within a class
- Values are accessed or modified through publically accessible methods

- Within a student class:
  - We might have private data fields: name, ID, gpa and program
  - And public setters and getters: getProgram, setProgram, etc.

- We have seen this all semester, but just haven't put a name to it
  - Encapsulation is not limited to Object-Oriented Programming, as it is present in some other programming paradigms, but it is considered one of the main pillars of OOP

1. Encapsulation

- Key takeaways:
  - Classes in Java allow us to bundle data fields and operations into a single unit
  - Encapsulation is used to hide the values of the data fields inside a class
  - Some data fields may be accessed or updated through public setters and getters

2. Abstraction

- During our **Interfaces** and **Abstract Data Type** (ADT) unit we discussed some of the benefits of hiding unnecessary details from clients/users:
- This allows us to manage the complexity of the project

- Clients/users should know what the system does and how to use it
  - but do not need to know the details about how it was implemented

2. Abstraction

- For example: Many people use mobile phones every day
- They know:
  - how to use the phone to initiate a voice call with someone
  - how to turn the volume up or down (or to vibrate)
  - how to use the camera to take pictures
- Most do ***not*** know
  - how the a wireless connection is established between caller and callee
  - how the buttons change the volume setting, or how the vibration works
  - how a digital image is created whenever they click the button to take a photo

2. Abstraction

- We can expand this notion when working on multiple components within the same software system as well

- **Encapsulation** allows a programmer to bundle all of the features of a component into a single unit, which we call a class in Java
- And allows the programmer to restrict access to fields that do not need to be accessed or manipulated externally, while still providing access to certain methods

- This can also apply to hiding unnecessary details from other programmers, even within the same team!

Abstraction Example: Mobile phone

- Let's say we are creating an app for a mobile phone
  - and the app uses the camera and vibration features

- Typically, there will be frameworks that allow to call methods to perform these types of operations for us:
  - a ***takePhoto*** method that returns image data
  - a ***vibrate*** method that we give a duration to that makes the device vibrate

- We don't want to have to write code to implement these things, we just want to use these operations in our app
  - and we probably don't really care how exactly this code was implemented

2. Abstraction

- Classes often provide an abstraction that hide internal implementation details (which are often unnecessary)
- When working with another class, all a programmer needs to know is which methods are available to call, and what input parameters need to be given in order to trigger their intended behaviour or result
- Programmers don't need to know how each method is implemented

- This makes large systems that have many different programmers working on many different components much easier to use.
  - It would be impossible for each program to learn all of the details of every other component, the design of the system would never get completed!

2. Abstraction

- Key takeaways
  - Abstraction is used to reduce the complexity of a system
  - This is achieved by hiding unnecessary details about how an operation works

3. Inheritance

- **Inheritance** allows us to make the information in our projects much more organized and manageable
  - subclasses inherit properties and behaviours from their parent class, allowing us to **reuse** existing code instead of writing new code for each class
  - this can make it easier to implement new classes:
    - the inherited features should already be testing and working correctly
    - focus is shifted to updating behaviours and adding new features applicable to the subclass
- By default, a subclass inherits all of the methods from its parent class
  - Subclasses can also **override** methods of its superclass to refine or change the behaviour for all instances of the subclass

3. Inheritance

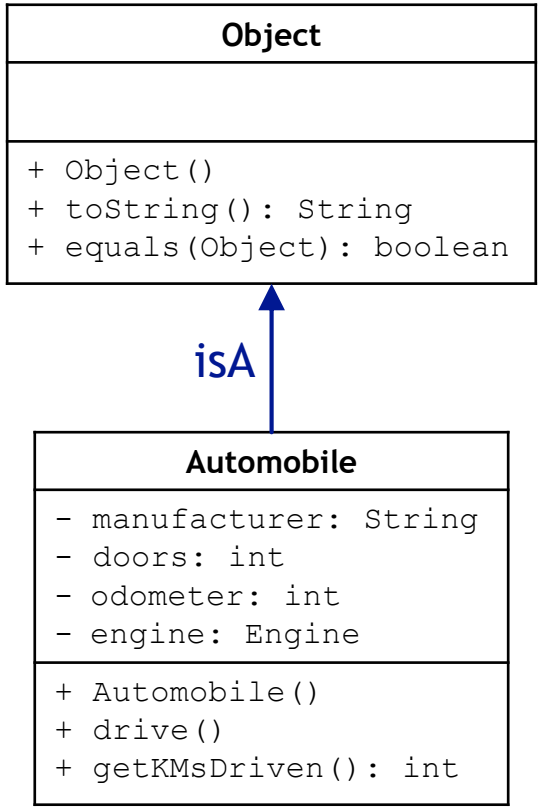
- There is some terminology commonly associated inheritance

- The **is-a** relationship:
  - When a class extends another class, there is an "**is-a**" relationship
  - A dog **is-a** mammal, a mammal **is-a** animal

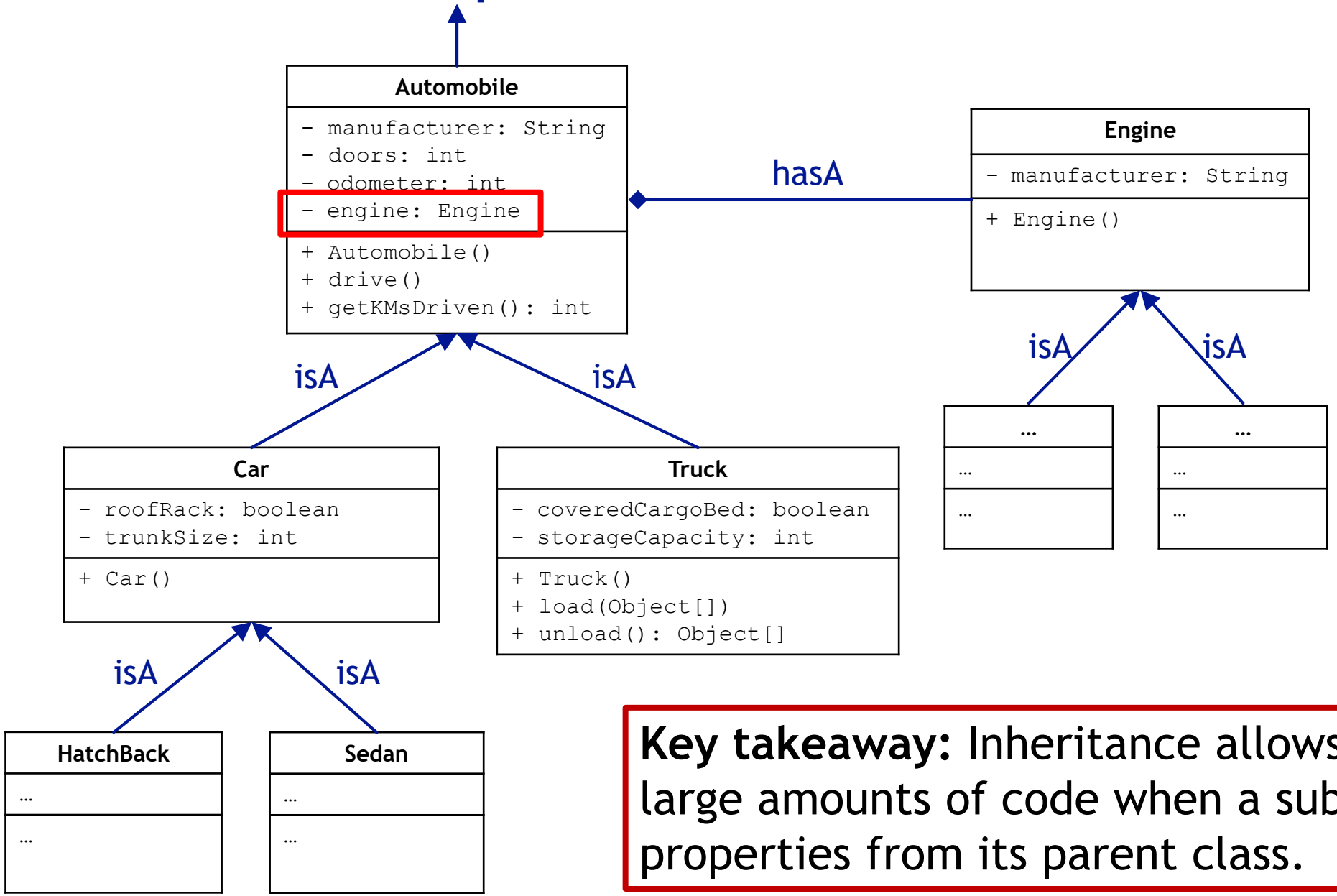
- There are also **has-a** relationships
  - This is related to **composition** instead of inheritance, as sometimes our objects might be **composed** of other objects
  - The fields in our objects might be other objects we have created

Inheritance example

- All of the classes we write implicitly extend Java's object class
  - https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html



Inheritance example



Key takeaway: Inheritance allows us to **reuse** large amounts of code when a subclass inherits properties from its parent class.

4. Polymorphism

- **Polymorphism**, which literally translates to *many forms*, at its core allows us to perform a single action in different ways.

- Something can have many forms in the real world too:
  - One person can be a mother, a partner, an employee, and a teammate.
  - One person might have different behaviours in different situations

- We can translate this idea into programming if we imagine that an object might have different behaviours or implementations in different situations
  - We can do this when we implementing interfaces and extending classes

4. Polymorphism

- Imagine we have implemented a list with an array or a linked list
- Assume both implementations implement a List interface
- We can create a reference variable of type List

```
List myList;
```

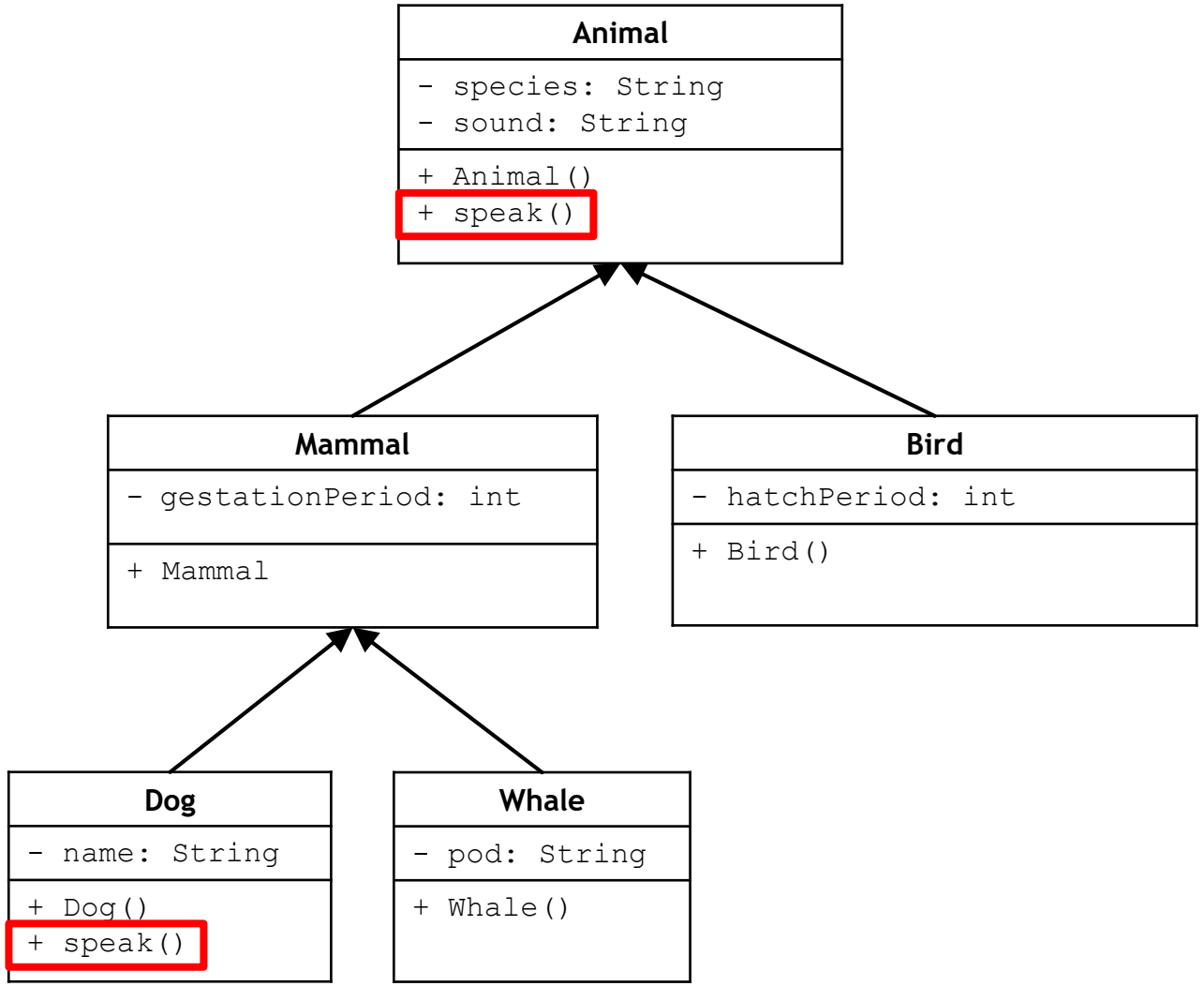
- And use either implementation:

```
myList = new ArrayList();
myList = new LinkedList();
```

- Both implementations implement the same List interface, so the same operations can be applied to `myList` (add, remove, get, etc)
  - But they are implemented in very different ways

4. Polymorphism

- We can also use polymorphism to produce different behaviours



4. Polymorphism

Animal speak() method:

```
public void speak() {
  System.out.println("I am a " + species + " and I say " +sound);
}
```

Dog speak() method (which overrides the Animal speak for dogs):

```
public void speak() {
  System.out.println("My name is "+name);
  super.speak();
}
```

4. Polymorphism

```
Animal[] pets = new Animal[5];
pets[0] = new Animal("lion", "roar");
pets[1] = new Mammal("pig", "oink", 4);
pets[2] = new Dog("Chauncy", "chihuahua", "yap yap");
pets[3] = new Whale("killer whale", 15);
pets[4] = new Mammal("cow", "moo", 9);

for (int i = 0; i < pets.length; i++) {
  pets[i].speak();
}
```

Animal speak()

Animal speak()

Dog speak()

Animal speak()

Animal speak()

Key takeaway: Polymorphism allows us to use an instance of a class as if it were different types. The method that is invoked is not determined at runtime