INHERITANCE

CS302 – Introduction to Programming
University of Wisconsin – Madison
Lecture 24

By Matthew Bernstein – matthewb@cs.wisc.edu

Inheritance

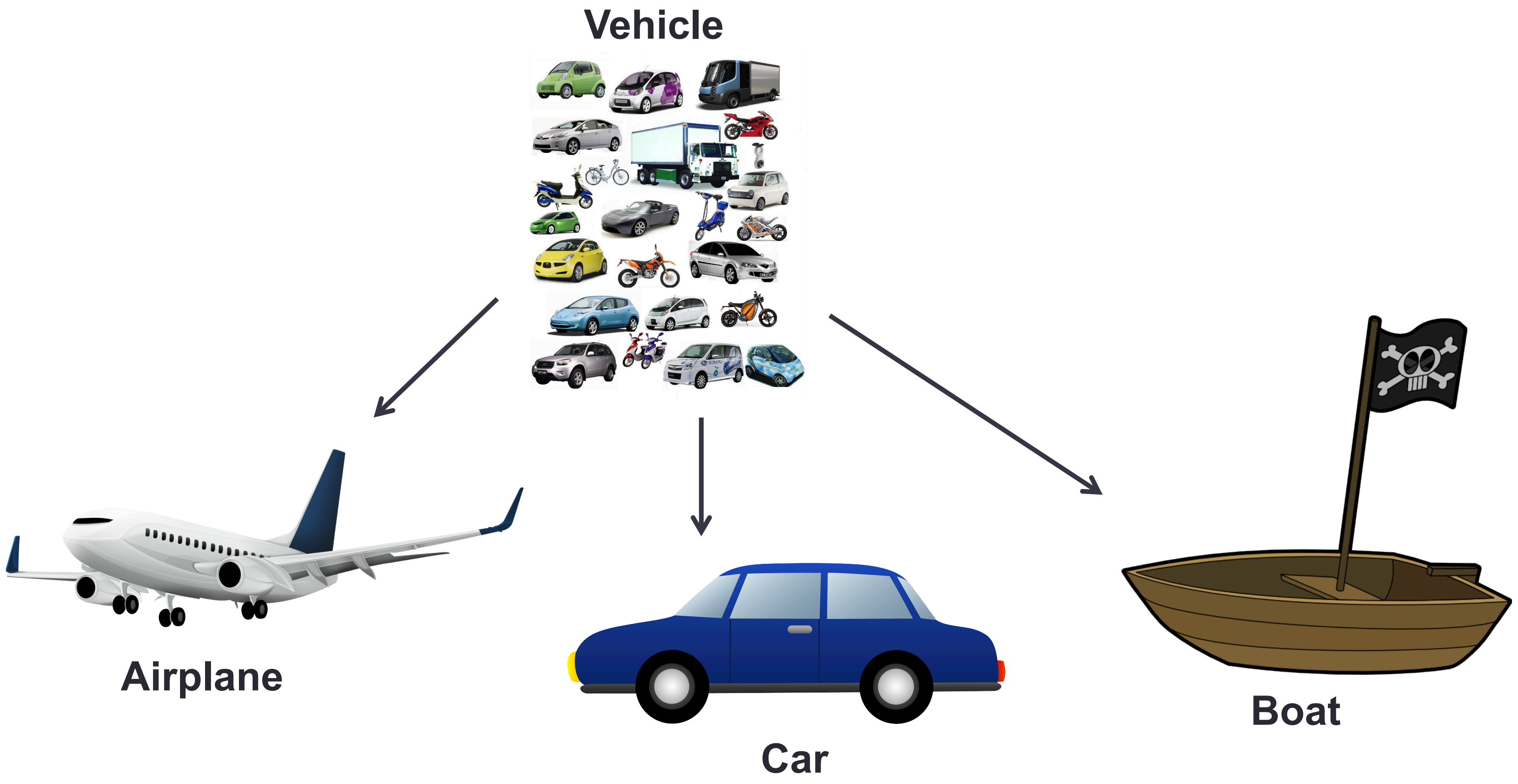
- In object-oriented programming, **inheritance** is a relationship between a more general class (called the **superclass**) and a more specialized class (called the **subclass**).
- The subclass inherits data and behavior from the superclass
- Inheritance allows us to represent an “is a” relationship between two class

“Is a” Relationship

- For example, suppose we have a program that uses the classes **Car**, **Airplane**, and **Boat**
- What do these classes have in common?
  - They are all vehicles
- A Car **“is a”** Vehicle
- We can therefore make a superclass called **Vehicle** and make the classes **Car**, **Airplane**, and **Boat** to be subclasses of **Vehicle**

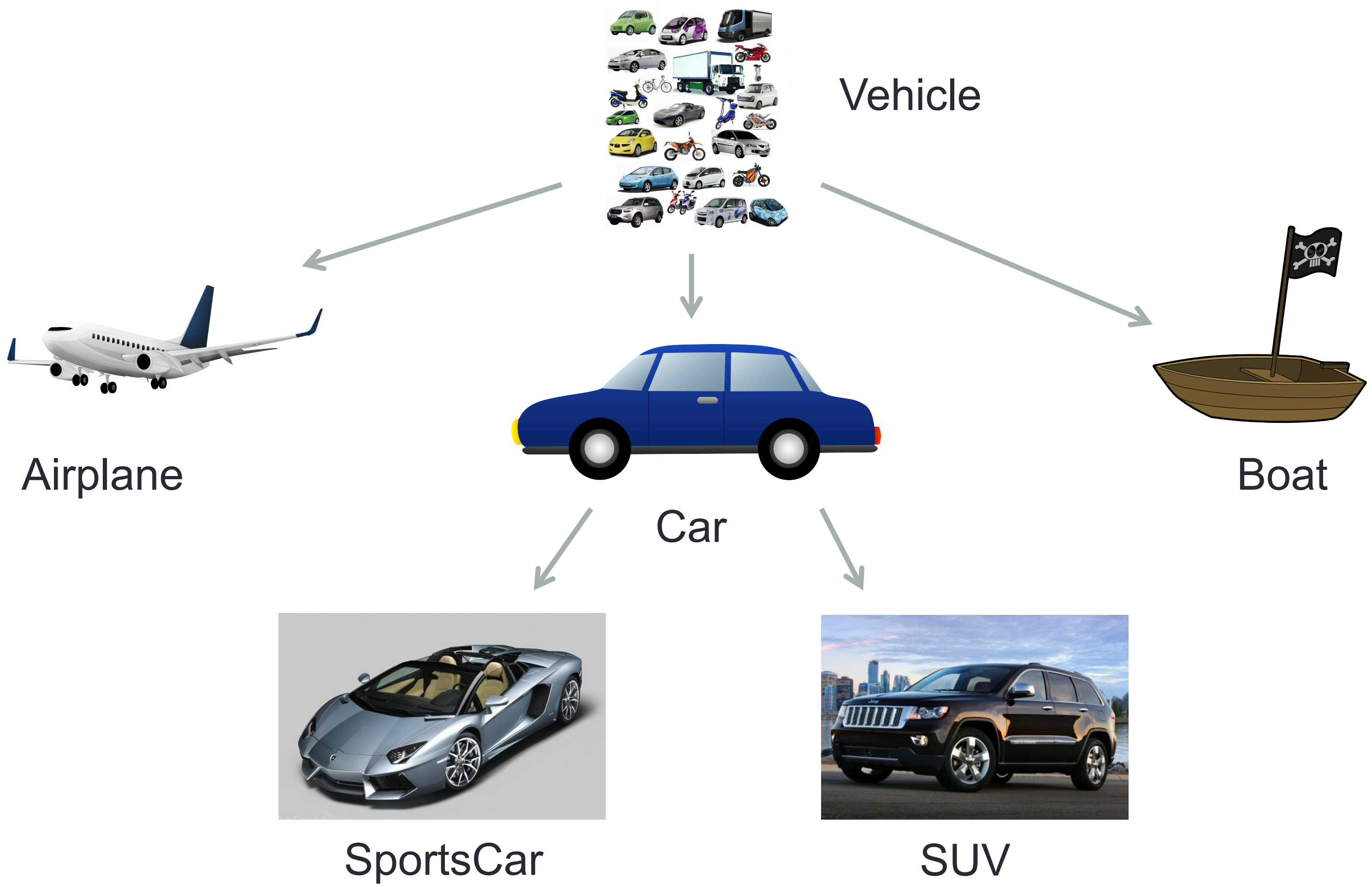
Inheritance Hierarchy

- This yields the following **inheritance hierarchy**:



Adding to the hierarchy

- Now let’s make subclasses of the Car class: **SportsCar**, **SUV**



The **Substitution Principle**

- The **substitution principle** states that you can always use a subclass object when a superclass object is expected
- For example, if we have a method that expects a Vehicle object:

```
void processVehicle(Vehicle v);
```

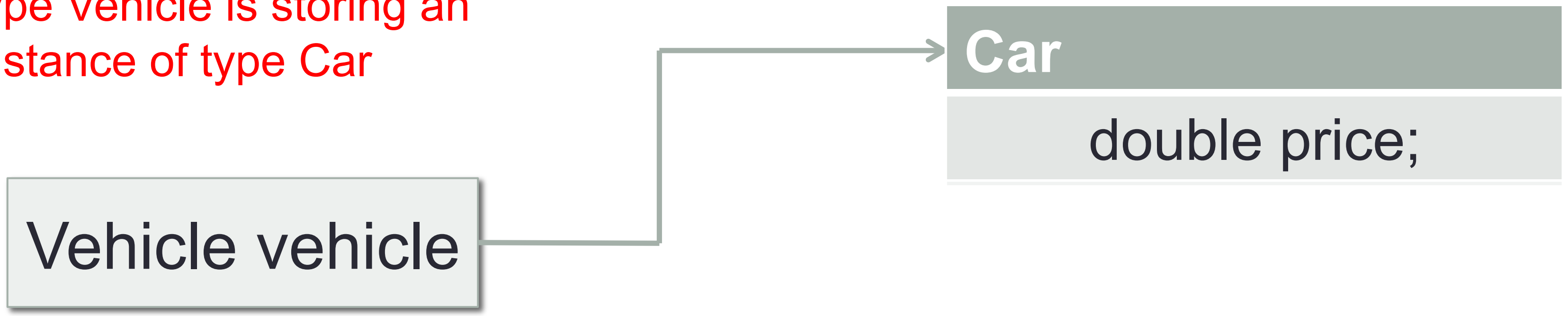
- We can pass a Car object because Car is a subclass of Vehicle
- Furthermore, we pass this method any subclass of Car such as SUV because SUV is a subclass of Car which is a subclass of Vehicle

Example of the Substitution Principle

- You can use a reference variable of a certain class to reference an object of this class's subclass:
- Example:

```
Vehicle vehicle = new Car();
```

A reference variable of type Vehicle is storing an instance of type Car



Inheriting Public Variables

- A subclass inherits all of the public instance variables from its superclass
- So for example, if we have a Vehicle class that has the following public variable:

```
public double topSpeed;
```

- All subclasses of vehicle will also have this. In our example, all classes below Vehicle in the inheritance hierarchy will also have the variable "topSpeed" by default even if this variable is not declared in the subclass itself

Inheriting Public Methods

- A subclass inherits all of the public methods from its superclass
- So for example, if we have a Vehicle class that has the following public method:

```
public void drive()
```

- You can call drive on an object of any subclass of Vehicle even though the drive() method is not defined or implemented inside the subclass itself

Declaring a Subclass

- You declare a class to be a subclass of another class using the reserved word **extends**
- Example:

```
public class Car extends Vehicle
{
                ...
}
```

This denotes that the Car class inherits from the Vehicle class

You Don’t Declare a Superclass

- Any class can be a super class, we don’t have to declare any class as a super class
- Thus you can decide to create a subclass of any class you wish

Inheriting from **Object**

- In Java, all classes inherit from a class called **Object**
- Object is considered the "cosmic" superclass of all class in Java
- You do not need to explicitly inherit from Object when defining your classes. This happens automatically.

Methods Inherited from **Object**

- Some useful methods you inherit from object includes:

  - **toString()** → returns a String representation of the Object
  - **hashCode()** → return this object's hash code
  - **equals()** → compare this object to another object
  - **clone()** → return an identical copy of this object

It is common to override some of these methods inside your class!

Calling a superclass's constructor from a subclass

- How does a subclass initialize its instance variables? More specifically, how can a subclass initialize the private variables in its superclass?
- We need to call the superclass's constructor from the subclass's constructor
- We use the Java reserved word **super** for referring to its superclass (this is similar to the **this** reserved word)
- Example:

```
…

super( );
…
```

Pass arguments to the superclass's constructor as required by the superclass

Why Inheritance?

- Inheritance increases code reuse and decrease redundant code
- Why is redundant code bad?
  - Obviously it takes longer to code when you need to repeat yourself
  - If you ever need to make a change in one section of your code, you will have to also change it in all of the redundant sections. This greatly increases the chances of causing bugs

Programming Exercise - Trivia

- Let’s write a program that will play a trivia game
- Our program will do something like the following:

```
In which country was the inventor of Java born?
1. Australia
2. Canada
3. Denmark
4. United States

2
Correct!
```

Quick Aside – Who invented Java?

This dude $\longrightarrow$

James Gosling

Superclass: **Question**

- We will make a superclass called Question from which other questions types will inherit (i.e. true or false, multiple choice, short answer, etc.)

```
class Question
{
        String question;
        String answer;

        // We'll fill in the rest as we go
}
```

Subclass: **MultipleChoiceQuestion**

- We'll make a class that inherits from **Question** called **MultipleChoiceQuestion**

```
class MultipleChoiceQuestion extends Question
{
        ArrayList<String> answerChoices;
        int correctChoice;

        // Let's fill in the rest as a class exercise
}
```

Our Class Hierarchy

- So far, our class hierarchy looks as follows:

$$\begin{tikzcd}
\text{Question} \arrow[d] \\
\text{MultipleChoiceQuestion}
\end{tikzcd}$$

This class contains all of the basic functionality needed in a Question

The subclasses implement all
Of the extended functionality necessary
To implement more specific question types

Question's Constructor

- Let's define a constructor for our Question superclass (nothing new here)

```
…

public Question(String question, String answer)
{
        this.question = question;
        this.answer = answer;
}

…
```

MultipleChoiceQuestion’s Constructor

- Using the **super** keyword, we would need to call **Question**’s constructor from **MultipleChoiceQuestion**’s constructor:

```
                    …

public MultipleChoiceQuestion(String question,
                              String answer)
{
      super(question, answer);

}
                    …
```

New access modifier: **protected**

- Remember that an access modifier determines where a class’s variables and methods are visible (i.e. can be accessed)
- **public** → Visible inside the class or outside its class (accessible from anywhere)
- **private** → Visible ***ONLY*** inside its class
- **protected** →(new) Visible inside its class, inside its package, ***OR*** inside its subclasses.

Overriding Methods in a Subclass

- Let’s say that our Question class has the following method for displaying its question to the user:

```
public void display()
{
        System.out.println( this.question );
}
```

This method simply prints its question String to the console

Overriding Methods in a Subclass

- However, we want objects of the MultipleChoiceQuestion subclass to also display all of the choices underneath the question. How do we do this?
- We **override** the superclass Question's display method
- How do we override a superclass's method?
- We define an identical method header in the subclass but implement the method differently.

Overriding Methods in a Subclass

- We override the "display" method inside the MultipleChoiceQuestion class:

It is best practice to always include the '@Override' annotation whenever you override a superclass's method

Print all of the choices under the question

```
@Override
public void display()
{
        System.out.println( this.question );

        for (int i = 0; i < choices.size(); i++)
        {
                System.out.println(i + ". " + choices.get(i));
        }
}
```

Overriding Methods in a Subclass

- Now, whenever we call a MultipleChoiceQuestion object’s display method, we will call the MultipleChoiceQuestion.display method instead of the Question.display method

instanceof

- You can check whether an object being referenced by a variable corresponding to the superclass is actually an instance of a subclass
- You use the **instanceof** operator
- This operator requires two operands (a reference variable and a type)
- It returns true if the first operand is a subclass of the second operand:

```
object_1 instanceof class
```

Example using **instanceof**

```
// Create question and answer Strings
String question = “What is your name?”;
String answer = “Joe Shmoe”;

// Create Question object
Question q = new Question(question, answer);

q instanceof Question;   // True

q instanceof MultipleChoiceQuestion;      // False
```

Example using **instanceof**

```
// Create question and answer Strings
String question = “What is your name?”;
String answer = “Joe Shmoe”;

// Create MultipleChoiceQuestion object
MultipleChoiceQuestion q
              = new Question(question, answer);

q instanceof Question;   // True

q instanceof MultipleChoiceQuestion;     // True
```

Programming Exercise: Finish the Trivia Game

- Create a program that reads in questions and answers from an input file. Then generate a quiz by posing the questions in a random order to the user.
- In the input file, each line should begin with a character that denotes the type of data that line represents:
  - ‘**Q**’ denotes a normal question,
  - ‘**A**’ denotes an answer
  - ‘**M**’ denotes a multiple choice question.
  - ‘**C**’ denotes a choice corresponding to the previous multiple choice question
  - ‘**>**’ denotes a new question/choice/answer combo

Trivia Game – File Input Format

- Example of a file using the input we specified:

```
Q Where do you go to school?
A Wisconsin
>
M Where is the inventor of Java from?
C Canada
C America
C Denmark
C Australia
A 0
```

Question

Answer

Separates the previous question from the next one

Multiple choice question

Choices