



# Module 1-1 1

Inheritance

# Objectives

- Should be able to describe the purpose and use of **inheritance**
- Should be able to, given an existing set of classes, properly identify **subclasses** and **superclasses**
- Should be able to properly define and use superclasses and subclasses in an inheritance hierarchy

# Principles of Object-Oriented Programming (OOP)

- **Encapsulation** - the concept of hiding values or state of data within a class, limiting the points of access
- **Inheritance** - the practice of creating a hierarchy for classes in which descendants obtain the attributes and behaviors from other classes
- **Polymorphism** - the ability for our code to take on different forms
- **(Abstraction)** – extension of encapsulation. We can't build a car from scratch, but we know how to use (drive) it.

## Inheritance!

# Inheritance

Real world objects can exhibit parent-child relationships. Consider the following examples:



- Humans, dogs, elephants, and whales are clearly quite different from each other, but they have enough similarities prompting biologists to classify them as mammals.
- Cars, motorcycles, and trucks are all motor vehicles, but they each have sufficient differences for the Department of Motor Vehicles to regulate them differently.
- In finance, the word account can refer to a checking account, a savings account, or mutual fund, but they all share similarities like a monthly balance and account holder name.

# Inheritance

Java and most modern programming languages provide powerful tools that enable developers to model these parent-child relationships.

- More specifically in Java one class can be classified as a child of a parent class.
  - The child class can inherit properties and methods defined by the parent.
- Inheritance can take on several forms:
  - A **concrete class** (all the classes we have seen so far) inheriting from another concrete class.
  - A concrete class inheriting from an **abstract class**.
  - A concrete class inheriting from an **Interface**.

# Inheritance

Described as:

- Parent – child relationship
- Superclass – subclass relationship
- Base – derived relationship
  
- A race-horse **is a** type horse
- A savings account **is a** type of bank account
- A tarantula **is a** type of spider



Described as an “is-a” relationship – goes one-way, not all spiders are tarantulas.



# Inheritance: Declaration

In this module, we will explore the situation where two concrete classes have a parent child relationship. A child class that will inherit from a parent must be defined following this syntax:

```
public class <<Name of Child Class>> extends <<Name of Parent Class>> {  
  
... // rest of your class declaration  
  
}
```



# Inheritance Example

Vehicle has defined several methods and data members. In this example, Vehicle serves as the parent class.

```
package te.mobility;

public class Vehicle {

    private int numberOfWheels = 4;
    private double engineSize;
    private String bodyColor;

    public int getNumberOfWheels() {
        return numberOfWheels;
    }
    public void setNumberOfWheels(int numberOfWheel
        this.numberOfWheels =
        numberOfWheels;
    }
}
```

We use the **super** keyword to refer to the parent's members and variables.

Car is a child class of Vehicle.. Note how it is able to call Vehicle's methods. The extends syntax is used to create this relationship.

```
package te.mobility;

public class Car extends Vehicle {

    public void report() {
        System.out.println(super.getNumberOfWheels());
        // 0, inherited from parent class
        // default value for integers.

        super.setNumberOfWheels(4);
        // we are calling the setter
        its parent

        System.out.println(super.getNumberOfWheels());
        // 4
    }
}
```

# Inheritance Example

Here we define another child class of Vehicle called Truck.

```
package te.mobility;

public class Truck extends Vehicle {

    public void report() {
        super.setNumberOfWheels(10);
        // we are calling the setter defined on
its parent
    }

    public void coupleCargoContainer() {
        System.out.println("coupling cargo
container");
        super.setNumberOfWheels(18);
    }
}
```

We now have a parent class with 2 child classes. Vehicle is the parent class. Both Truck and Car extends from vehicle, making them child classes of the vehicle.

The Truck class has its own unique method, it has a method called coupleCargoContainer() which is unique to the Truck class, and not part of the Vehicle or Car class.

# Inheritance Example

```
public class GarageDemo {  
  
    public static void main(String args[]) {  
  
        Car myCar = new Car();  
        myCar.setup();  
  
        System.out.println(myCar.getNumberOfWheels());  
  
        Truck myTruck = new Truck();  
        myTruck.setup();  
  
        System.out.println(myTruck.getNumberOfWheels());  
        myTruck.coupleCargoContainer();  
  
        System.out.println(myTruck.getNumberOfWheels());  
  
        // This is an invalid call:  
        //myCar.coupleCargoContainer();  
  
    }  
}
```

Suppose have an orchestrator class called **GarageDemo** with a main method that will instantiating new cars and trucks based on the setup we've established so far.

Output will be 4

Output will be 10

Output will be 18

This is an invalid statement, the `coupleCargoContainer` method is unique to the **Truck** class.

# Effect of Private Modifiers on Inheritance

The access modifiers present on the parent class' data members is not trivial.

- Data members and methods marked as **private** on a parent class cannot be inherited by a child class.
- Data members and methods marked as **protected** can be inherited by a child class even if it's on a different package.  
\*\*\*\*\* Limit the accessibility as much as possible \*\*\*\*\*
- Use private, unless you must use protected. Use protected, unless you must use public. Never use public on an instance variable!

# Effect of Private Modifiers on Inheritance


Consider the following example:

```
public class Vehicle {  
    ...  
    private String privateMethod() {  
        return "private";  
    }  
    ...  
}
```

We are assuming that the Car class extends from Vehicle like on the previous examples.

```
public class GarageDemo {  
    public static void main(String  
    args[]) {  
        Car myCar = new  
        Car();  
        myCar.setup();  
        myCar.privateMethod();  
        ...  
    }  
}
```

This is an  
invalid call.



# Constructors with inheritance

```
public class Vehicle {  
    ...  
    public Vehicle() {  
        System.out.println("Vehicle");  
    }  
    ...  
}
```

```
public class Truck extends Vehicle {  
    ...  
    public Truck() {  
        System.out.println("Truck");  
    }  
    ...  
}
```

```
public class GarageDemo {  
    public static void main(String args[]) {  
        Truck myTruck = new Truck();  
        ...  
    }  
}
```

Output will be:  
Vehicle  
Truck

# Constructors on Parent Classes

If a parent has implemented a constructor, a child class must add a call using `super(...)` to invoke the parent's constructor with the correct arguments.

The syntax of `super(...)` is as follows:

```
public ChildClass(<<argument 1>>, <<argument2>>, ....) {  
  
    super(<<argument1>>, <<argument2>>, ...);  
  
}
```

The arguments listed are arguments on the parent's constructors

# Constructors on Parent Classes: Example

```
package te.mobility;
```

```
public class Vehicle {
```

```
    private int numberOfWheels;
```

```
    private double engineSize;
```

```
    private String bodyColor;
```

```
    public Vehicle(int numberOfWheels, double engineSize, String bodyColor)  
{
```

```
        this.numberOfWheels = numberOfWheels;
```

```
        this.engineSize = engineSize;
```

```
        this.bodyColor = bodyColor;
```

```
    }
```

```
    ...
```

```
}
```

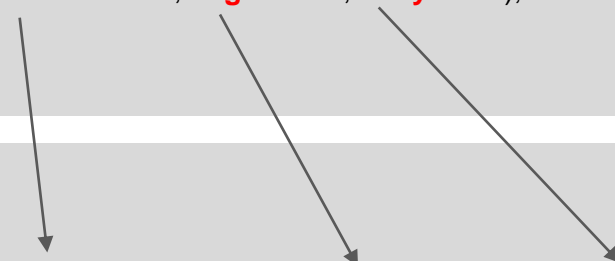
There is now a constructor in the parent Vehicle class.



# Constructors on Parent Classes: Example

Note how the child class, Truck will now have to implement a constructor with a `super(...)` call.

```
public class Truck extends Vehicle {  
  
    public Truck(int numberOfWheels, double engineSize, String bodyColor) {  
        super(numberOfWheels, engineSize, bodyColor);  
    }  
  
    ...  
}
```



```
public class Vehicle {  
    ...  
  
    public Vehicle(int numberOfWheels, double engineSize, String bodyColor)  
    {  
  
        this.numberOfWheels = numberOfWheels;  
        this.engineSize = engineSize;  
        this.bodyColor = bodyColor;  
  
    }  
    ...  
}
```

The `super(...)` call is a call to the parent constructor, providing any required parameters

# Constructors on Parent Classes: Example

In the Garage orchestrator class note how we are able to instantiate a new Truck with the constructor.

```
package te.main;

import te.mobility.Truck;

public class GarageDemo {

    public static void main(String args[]) {

        Truck cargoTruck = new Truck(10, 14.8, "red");

    }

}
```

# Multiple Constructors

Classes can contain more than one constructor, each taking a different number of arguments.

We are overloading the constructor (just like we can overload any other method)

## Constructor Overloading

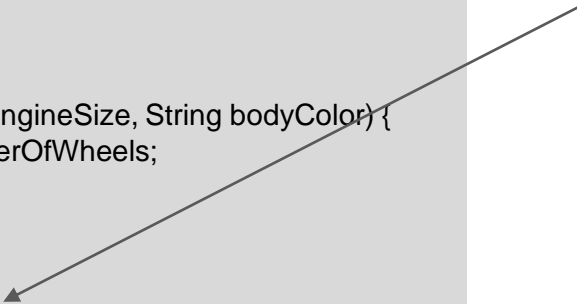
- 💡 Constructors can be overloaded:
  - An overloaded constructor provides **multiple ways to set up a new object**
  - The overloaded constructors differ by the *number* and *type* of parameters they get.
- 💡 When we construct an object, the compiler decides which constructor to invoke according to the type of the actual parameters
- 💡 A constructor with no parameters is called a **default constructor**



# Multiple Constructors Example

Consider the following example:

```
public class Vehicle {  
  
    private int numberOfWheels;  
    private double engineSize;  
    private String bodyColor;  
  
    public Vehicle(int numberOfWheels, double engineSize, String bodyColor) {  
        this.numberOfWheels = numberOfWheels;  
        this.engineSize = engineSize;  
        this.bodyColor = bodyColor;  
    }  
  
    public Vehicle(int numberOfWheels, double engineSize) {  
        this.numberOfWheels = numberOfWheels;  
        this.engineSize = engineSize;  
    }  
  
    ...  
}
```




Note that there is now a second constructor that does not take a bodyColor argument.

# Multiple Constructors Example

```
public class Truck extends Vehicle {  
  
    public Truck(int numberOfWheels, double engineSize, String bodyColor) {  
        super(numberOfWheels, engineSize, bodyColor);  
    }  
  
    public Truck(int numberOfWheels, double engineSize) {  
        super (numberOfWheels, engineSize);  
    }  
}
```

Note how the child class has also implemented a matching second constructor and called the 2 argument parent constructor using super.



# Objectives

- Should be able to describe the purpose and use of **inheritance**



# Objectives

- Should be able to describe the purpose and use of **inheritance**
- Should be able to, given an existing set of classes, properly identify **subclasses** and **superclasses**

```
class Data {  
    private int data1;  
    private int data2;  
  
    void setData(int da1,int da2)  
    {  
        data1 = da1;  
        data2 = da2;  
    }  
  
    int getData1()  
    {  
        return data1;  
    }  
  
    int getData2()  
    {  
        return data2;  
    }  
}  
  
class NewData extends Data{  
    int data3;  
    int data4;  
  
    void setNewData(int da1,int da2,int da3,int da4)  
    {  
        setData(da1,da2);  
        data3 = da3;  
        data4 = da4;  
    }  
  
    void showNewData()  
    {  
        System.out.println("data1 = "+getData1());  
        System.out.println("data2 = "+getData2());  
        System.out.println("data3 = "+data3);  
        System.out.println("data4 = "+data4);  
    }  
}  
  
public class Javaapp {  
    public static void main(String[] args) {  
        NewData obj = new NewData();  
        obj.setNewData(20, 40, 60, 80);  
        obj.showNewData();  
    }  
}
```

# Objectives

- Should be able to describe the purpose and use of **inheritance**
- Should be able to, given an existing set of classes, properly identify **subclasses** and **superclasses**
- Should be able to properly define and use superclasses and subclasses in an inheritance hierarchy

## Inheritance in Java

