Unit 9: Inheritance
Introduction to Inheritance

Adapted from:
1) Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp
2) Runestone CSAwesome Curriculum

https://longbaonguyen.github.io
Inheritance

You may have heard of someone coming into an inheritance, which often means they were left something from a relative who died. Or, you might hear someone say that they have inherited musical ability from a parent.

In Java all classes can inherit attributes (instance variables) and behaviors (methods) from another class.

The class being inherited from is called the parent class or superclass. The class that is inheriting is called the child class or subclass.
Inheritance

**inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
- a way to group related classes
- a way to share code between two or more classes

One class can *extend* another, absorbing its data/behavior.
- **superclass**: The parent class that is being extended.
- **subclass**: The child class that extends the superclass and inherits its behavior.
  - Subclass gets a copy of every instance variable and method from superclass
Syntax

A class can extend another class by using the keyword `extends` then the name of the class it is extending.

Below, Employee extends Person.

```java
public class Person{
    private String name;
    public Person(String theName){
        this.name = theName;
    }
    ...
}

public class Employee extends Person{
    // not shown
}
```
Inheritance

When one class inherits from another, we can say that it is the *same kind of thing* as the **parent class** (the class it inherits from).

For example, a car is a kind of vehicle. This is sometimes called the *is-a* relationship, but more accurately it’s a *is-a kind of* relationship.

A motorcycle is a vehicle. Or a motorcycle is a kind of vehicle. All vehicles have a make, model, and year that they were created. All vehicles can go forward, backward, turn left and turn right.
A **UML (Unified Modeling Language) class diagram** shows classes and the relationships between the classes as seen in Figure 1. An open triangle points to the parent class. The parent class for Car and Motorcycle is Vehicle. The Vehicle class has two child classes or subclasses: Car and Motorcycle.
Inheritance allows you to reuse data and behavior from the parent class.

If you notice that several classes share the same data and/or behavior, you can pull that out into a parent class. This is called **generalization**.

For example, if you wrote two classes Customers and Employees and both of which have instance variables `name` and `address`, then it makes sense write the general Person class with the variables `name` and `address` and have Customers and Employees inherit from Person.

This allow us to **reuse** code from Person. See UML on the next page.
Figure 2: A UML Class Diagram Showing Inheritance
Conversely, inheritance is also useful for **specialization** which is when you want most of the behavior of a parent class, but want to do at least one thing differently and/or add more data.

The previous example below can also be seen as specialization. Suppose that you already wrote the Person class and you want a Customer class that has all of the attributes of Person but also has a credit card. Or you want an Employee class that has an additional instance variable for id.
Subclasses **inherit all the public and private instance variables** in a superclass that they extend, but they **cannot directly access private variables**.

And constructors are **not inherited**.

How do you initialize inherited private variables if you don’t have direct access to them in the subclass?

In Java, the superclass constructor can be called from the first line of a subclass constructor by using the keyword **super** and passing appropriate parameters. This allows the parent constructors to initialize the variables.
Below is the Person class. We will create an Employee class which extends this class.

class Person{
    private String name;
    public Person(String theName){
        this.name = theName;
    }
    public String getName(){
        return name;
    }
    // other methods not shown
}
The subclass constructor cannot directly initialize the super class' private variables.

```java
public class Employee extends Person{
    private int id;
    public Employee(String name, int id){
        this.name = name;  // error, name is private
        this.id = id;
    }
    public int getId(){
        return id;
    }
    // other methods not shown
}
```
The superclass constructor can be called from the first line of a subclass constructor by using the keyword `super` and passing appropriate parameters.

`super()` **must be called on the first line of the constructor!**

```java
class Employee extends Person{
    private int id;
    public Employee(String name, int id){
        super(name);  // call Person's constructor
        this.id = id; // to initialize name variable
    }
    // other methods not shown
}
```

Now each Employee object has an id variable in addition to the inherited name variable.
public class Main{
    public static void main(String[] args){
        Person p = new Person("Mike");
        System.out.println(p.getName()); // Mike
        Employee emp = new Employee("Dani", 123);
        System.out.println(emp.getName()); // Dani
        System.out.println(emp.getId()); // 123

        Person p1 = new Employee("Mike", 11);
        // correct, every Employee is a Person.

        Employee p2 = new Person("Mike");
        // incorrect, not every Person is an Employee
    }
}
If a class has no constructor in Java, the compiler will add a no-argument constructor. A no-argument constructor is one that doesn’t have any parameters, for example public Person().

If a subclass has no call to a superclass constructor using super as the first line in a subclass constructor then the compiler will automatically add a super() call as the first line in a constructor.

So, be sure to provide no-argument constructors in parent classes or be sure to use an explicit call to super() as the first line in the constructors of subclasses.

See the next slide for an example of a common constructor error.
If a subclass has no call to a superclass constructor using `super` as the first line in a subclass constructor then the compiler will automatically add a `super()` call as the first line in a constructor.

**Find the error.**

```java
class Person{
    // Suppose this class has only this one constructor.
    public Person(String theName){...}
}

public class Employee extends Person{
    private int id;
    public Employee(String name, int id){
        this.id = id;
    }
}
```

implicitly call `super()` since subclass constructor has no `super()` call.

But parent class Person has no default constructor. Error!
Fix # 1

The simplest way to fix the previous error is to **always explicitly call super with the appropriate parameters** as the first line in every subclass constructor.

```java
class Person{
    // Suppose this class has only this one constructor.
    public Person(String theName){...}
}

public class Employee extends Person{
    private int id;
    public Employee(String name, int id){
        super(name);
        this.id = id;
    }
}
```

It is best to explicitly call `super()` with the appropriate parameters.
Fix # 2

Another way to fix the previous error is to provide a default (no argument) constructor in the parent class.

class Person{
  public Person(){…}
  public Person(String theName){…}
}

class Employee extends Person{
  private int id;
  public Employee(String name, int id){
    this.id = id;
  }
}

implicitly call super() since subclass constructor has no call. And since parent class does have a default constructor, no error!
Levels of Inheritance

Multiple levels of inheritance in a hierarchy are allowed.

```java
public class Person { // implementation not shown
}
public class Employee extends Person { // not shown
}
public class Lawyer extends Employee { // not shown
}

Find the error. Errors in red below.

```java
public class Main {
    public static void main(String[] args) {
        Person p1 = new Employee("Mike", 11);
        Employee p2 = new Lawyer("John", 22);
        Lawyer p3 = new Person("Katie");
        Lawyer p4 = new Employee("Jim", 33);
        Person p5 = new Lawyer("Michele", 44);
        Employee p6 = new Person("Jack");
    }
}
```
A subclass inherits all **public methods** from its superclass, and these methods remain public in the subclass. But, we also usually add more methods or instance variables to the subclass.

```java
class Person{
    ...
    public String getName(){...}
}
```

```java
public class Employee extends Person{
    ...
    public void printJobDescription(){...}
}
```

Employee inherits `getName()` from Person.

Employee cannot access name directly but can access it indirectly through `getName` and `setName`, etc...

`printJobDescription()` is a new method in Employee that is not in Person.
Overriding methods

A subclass inherits all public methods from its superclass, and these methods remain public in the subclass. But, we also usually add more methods or instance variables to the subclass.

Sometimes, we want to modify existing inherited methods.

**override**: To write a new version of a method in a subclass that replaces the superclass's version.

- To override an inherited method, the method in the child class must have the **same name, parameter list, and return type (or a subclass of the return type)** as the parent method. Any method that is called must be defined within its own class or its superclass.

**Have we done this before? Answer:** `toString()`
Overriding a method of the superclass.

```java
public class Employee extends Person{
    // some constructors and methods not shown
    public String getVacationForm() {
        return "pink";
    }
}

public class Lawyer extends Employee {
    // overrides getVacationForm method in Employee class
    public String getVacationForm() {
        return "yellow";
    }
    // Lawyer has new method sue()
    public void sue(){
        System.out.println("I'll see you in court!");
    }
}
```
A superclass reference can point to a subclass object. Calling a method from this reference always calls the overridden (if there is one) method.

```java
public class Main{
    public static void main(String[] args) {
        Employee emp = new Employee("Dani", 123);
        System.out.println(emp.getVacationForm()); // pink
        Employee sarah = new Lawyer("Sarah", 345);
        System.out.println(law.getVacationForm()); // yellow
    }
}
```

Note: sarah is an Employee reference but points to a Lawyer object. Calling getVacationForm on it actually calls the Lawyer's version NOT Employee's.
A superclass reference, however, cannot call a method of a subclass that is not inherited from the superclass. Doing so will cause a compiler-error.

```java
public class Main{
    public static void main(String[] args)
    {
        Employee sarah = new Lawyer("Sarah", 345);
        System.out.println(law.getVacationForm()); // yellow
        sarah.sue(); // compiler-error
    }
}
```

Note: Even though the sarah reference points to an actual Lawyer object. Because sarah is an Employee reference, it cannot call the sue() method which is a new method in Lawyer that is not in Employee. (More on this topic in the next lecture)
public class Main{
    public static void main(String[] args) {
        Employee emp = new Employee("Dani", 123);
        System.out.println(emp.getVacationForm()); // pink
        Employee law = new Lawyer("Sarah", 345);
        System.out.println(law.getVacationForm()); // yellow
        law.sue(); // error!
    }
}

Note: law is an Employee reference but points to a Lawyer object. Calling getVacationForm on it actually call the Lawyer's version NOT Employee's.
Another Example

public class Parent{
    public void method1(){
        System.out.println("Parent");
    }
}

public class Child extends Parent{
    public void method1(){
        System.out.println("Child");
        public void method2() { // implementation not shown }
    }
}

public class Main{
    public static void main(String[] args){
        Parent a = new Parent();
        Child b = new Child();
        Parent c = new Child();
        a.method1(); // Parent
        b.method1(); // Child
        c.method1(); // Child
        c.method2(); // error!
    }
}
Sometimes you want the subclass to do more than what a superclass’ method is doing. You want to still execute the superclass method, but you also want to override the method to do something else.

But, since you have overridden the parent method how can you still call it? You can use super.method() to force the parent’s method to be called.

We’ve used super() before to call the superclass’ constructor. There are two uses of the keyword super:

- **super();** or **super(arguments);** calls just the super constructor if put in as the first line of a subclass constructor.
- **super.method();** calls a superclass’ method (not constructors).
The keyword `super` is very useful in allowing us to first execute the superclass method and then add on to it in the subclass.

area of circle = $\pi \times \text{radius}^2$

area of cylinder = $2 \times \text{area of circle} + \text{lateral area}$

```java
class Circle{
    private double radius;
    public double getArea(){
        return Math.PI * radius * radius;
    }
}

class Cylinder extends Circle{
    double height;
    ...
    public double getArea(){
        return 2 * super.getArea() + 2 * Math.PI * getRadius() * height;
    }
}
```
Write the superclass Student and subclass GradStudent.

The Student class has a private string name and public integer id. It has a one constructor which accepts a string and an integer.

Student has getName(), setName (String n) and printWelcome() (prints welcoming message with their name for example "Welcome! Mike Smith!")
Write the subclass GradStudent.
The GradStudent class has a private string `dissertationTopic`. It has one constructor that accepts a String for name, another String for dissertation topic and one for id.
GradStudent has `getTopic()`, `setTopic(String t)`, `printWelcome()` public methods. `printWelcome()` overrides the same method from the superclass Student and prints for example "Welcome to Graduate School! Mike Smith!".

Write the driver class. Create at least one Student object and one GradStudent object using the Student reference for both. Print out welcoming messages for each. What do you notice?