Topics: Inheritance basics, overriding, dynamic dispatch
Ref: CSC20 PAL WS Classes and Objects

Now that we know how to write classes, we need a system design mechanism to allow different classes to share relationships and interact with one another.

Among human beings, every child inherits certain genetic characteristics of their parents, while also possessing the ability to gain new, unique skills and behaviors. For example, Mark inherits his family name, his eye color, and his empathetic nature from his father Sam. But Mark is also sporty and musically talented, unlike his father.

In the code below, we see that the Child class automatically possesses the class variables of the Father class without needing to explicitly declare them. It is also free to include additional class variables and methods not present in the Father class.

```
class Father {
     String family_name = "brown";
     String eye_color = "green";}

class Child extends Father {
     String instrument = "piano";
     String sport = "basketball";}

public class BasicInheritance {
     public static void main(String[] args) {

          Father Sam = new Father();
          System.out.println(Sam.family_name)
          System.out.println(Sam.eye_color)

          Child Mark  = new Child();
          System.out.println(Mark.family_name)
          System.out.println(Mark.eye_color)
          System.out.println(Mark.instrument)
          System.out.println(Mark.sport)
     }}
```

Inheritance reduces replicated code and allows child classes to creatively expand beyond the attributes of the parent class. For example, a father can

have many children who retain his family name, but play different sports which he does not.

Ex.1

1) Create two Java class called Daughter and Son which extend the 'Father' class: one for a daughter who can play hockey instead of basketball, and one for a son who goes swimming instead. The daughter can play the guitar while the son is not into music. Modify the BasicInheritance class to create a Son and Daughter object. Display all the characteristics of each object.
2) Change the attributes in the Father class to *private.* How can a child class access the private fields of the parent class? Make updates that would allow your program to work.
3) Add a constructor to the Father that takes in parameters for each of the fields. Look up the super keyword. Use it to create constructors for each of the child classes. Update the BasicInheritance class to create a Son and Daughter object with parameters. Display both objects.

Ex.2 Create Java classes to assist a Honda dealership promote characteristics of different models. Add constructors, private fields, getters and setters and a toString method that returns the string representation of each object. Write a driver that creates multiple Honda cars of various models and displays them. Do you see that inheritance is an "is a" relationship? E.g. A Civic "is a" Honda.

The dealership would first need a parent class called 'Honda', which is described by:

- Honda:
    - Brand: Honda (String type)
    - Automatic transmission: True (Boolean type)
    - Year: 2023 (integer type)

They also need the following child classes, each with their own unique characteristics:

- Civic:
    - Affordability Rating: 4.5 (double type)
    - Sedan: True (Boolean type)
- Pilot:

    - Fuel Tank: 20 gallons (integer type)
    - Horsepower: 280 (double type)
- Odyssey:
    - Seats: 8 (integer type)
    - Interior Size: maximum (String type)

We can also provide a child class with overriding methods to redefine and extend the basic methods of the parent class. This is important because when you have multiple unique child classes, they will need to implement the same inherited method in different ways.

Ex. 3 Write the Java method 'calculateVolume()' for the classes 'Cylinder' and 'Cone', which overrides the original method in the parent class 'CircularSolid'. The volume of a cylinder is $V = (\pi * \text{radius}^2 * \text{height})$, and the volume of a cone is $V = (\frac{1}{3} * \text{height} * \pi * \text{radius}^2)$.

Note that the name and signature of the overridden method must exactly match that of the parent.

```
class CircularSolid {
     double radius;
     double height;

     //parent or "super" constructor
     CircularSolid(double radius, double height){
          this.radius = radius;
          this.height = height;
     }

     double calculateVolume(){
          return 0; // since we do not know the type, we are
unable to use an appropriate formula.
     }
}

class Cylinder extends CircularSolid {

     // child constructor
     Cylinder(radius, height){
          super(radius, height); // call parent constructor
```

```
    }

    // overridden method
    double calculateVolume(){
        //fill in using formula for volume of cylinder
    }
}


class Cone extends CircularSolid {

    // overridden constructor
    Cone(radius, height){
        super(radius, height); // call parent constructor
    }

    // overridden method
    double calculateVolume(){
        // fill in using formula for volume of cone.
    }

}
```

Ex4. Write a toString method for the CircularSolid class as follows:

```
    public String toString() { // for the CircularSolid class
        return "I am a Circular Solid – my volume is "+
    calculateVolume();}
```

Create a Cone object in the driver as follows:

```
Cone c = new Cone(5,10);
Call c.toString();
```

Observe the output. Is the calculated volume 0 or 523.333? Why is this so? Read up on dynamic dispatch. What is the sequence of method calls here?

Ex5. Write a toString method for the Cone class. Observe the output when it is called on the `Cone c` object as in the exercise above. How can you change the code to observe the exact same output as Ex 5?