**Week 9 Lab – Inheritance**

# Lab Intro

In the lecture this week, we saw a very powerful object-oriented concept: **inheritance**. In this lab, we will implement programs that make use of inheritance in order to reduce code duplication across classes.

**Key Term – Inheritance**Inheritance allows one class to inherit the attributes and methods of another class. This relationship is referred to as an “is-a” relationship (e.g. Dog is-a Animal). This is achieved in Java using the **extends** keyword. E.g.   
class Dog extends Animal, where Dog is the subclass and Animal is the superclass.



After this session, you should have experience of implementing inheritance that can be directly applied to your assessment. **Remember**: most of the concepts we see in IntelliJ are applicable in **Processing**. Since week 1 we have been learning the syntax of **Java** – IntelliJ and Processing are merely tools that have helped us to learn **Java** concepts.

Launch IntelliJ and create a new **Project** named **Week 9**.

## Learning Outcomes

* **Identify** suitable use cases for inheritance
* **Implement** programs that utilise inheritance to reduce code duplication across classes

## Resources

* Week 9 lecture slides
* Week 9 reference sheet

# Exercise 1 – Recognising “is-a” Relationships



Before you start coding anything in this lab, look at the “is-a” (extends) **relationships** in Table 1 and highlight those that are valid in **green**, and those that are not suitable “is-a” relationships in **red**. A couple of these have been done for you to get started.

**Remember!**Remember the “is-a” relationship test. A cat is-a animal, but an animal is not a cat. A car is-a vehicle, but a vehicle is not a car.

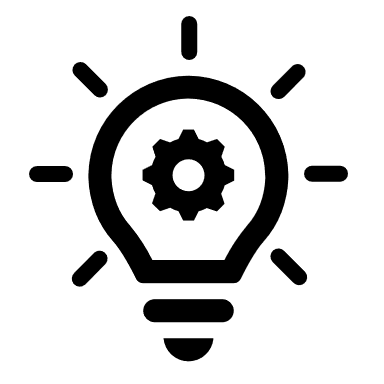


Table Example Inheritance Relationships

|  |  |  |
| --- | --- | --- |
| Person extends Athlete | Dog extends Animal | Bow extends Weapon |
| Building extends House | Container extends Box | Vehicle extends Bicycle |
| Biro extends Pen | Room extends Kitchen | Plane extends Vehicle |
| Browser extends Chrome | Animal extends Bird | Silk extends Material |

# Exercise 2 – Implementing an Inheritance Hierarchy from UML



Let’s start by creating a simple inheritance hierarchy (depicted in the **UML** diagram in Figure 1) involving three classes – **Baker** and **Builder** will be our subclasses, which will extend the **TradesPerson** class (the superclass).

**Key Term – UML  
UML** refers to the **Unified Modelling Language** – a way of graphically representing an object-oriented system. There are many types of UML diagrams, with the most popular being a **class diagram**, which allows us to quickly see the name, attributes, and methods of a class.



1. Create the **TradesPerson** class (**note**: this class will **not** have a main method, it will just be used to represent a TradesPerson data type).
   1. Look at the UML class diagram for the **TradesPerson** class (Figure 1) and **add the member variables** you see in the middle compartment.
   2. Add a **constructor** to your **TradesPerson** class that accepts the three variables to initialise the three members (name refers to the persons name). See the **information** box on how to quickly generate a constructor using the IntelliJ shortcut.
   3. Implement the display method, which shoud print to console the member values.
2. Create the **Baker** class and make it extend the **TradesPerson** class using the *extends* keyword (again, no main method for this class).
   1. You will notice an **error** in the code – you can resolve this by adding a constructor in this subclass to initialise **name**, **wage** and **speciality** (this member is specific to bakers). Your constructor must call the superclass’ constructor (see **slides** if you need to see an example of this). Your constructor should call the superclass’ constructor (super keyword) to deal with **name**, pass **“baker”** for the trade, and **wage**.
   2. Add a method to your **Baker** class named **makes** (no return type – so void). The **makes** method can print <bakers name> " bakes breads and cakes." and the display the speciality.

**Key Term – Method Overriding**A subclass can **override** a method it has inherited from its superclass and provide its own implementation for that method.



* 1. **override** the **display** method inherited from the TradesPerson class so that the display method display all information about a baker and calls the **makes** method. See slides 21-23 showed examples of method overriding**.**

|  |
| --- |
| Builder |
| tools: String[ ] |
| display() : void  uses() : String |

|  |
| --- |
| Baker |
| speciality: String |
| display() : void |
| makes() : String |

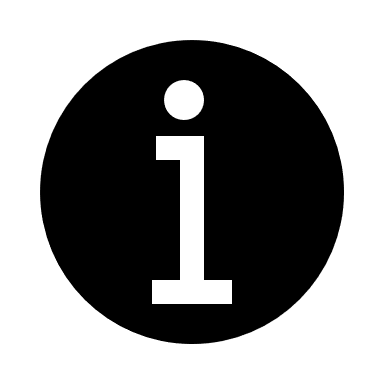
|  |
| --- |
| TradesPerson |
| name : String |
| trade : String |
| wage : int |
| display() : void |

**Information - IntelliJ Shortcut**Holding **ALT** and pressing the **Insert** key whilst your cursor is in a class allows you to generate a constructor and select which fields to initialiseGraphical user interface, text, application, chat or text message

Description automatically generated

Graphical user interface, text, application

Description automatically generated



1. Create a class named **InheritanceDemo** with a **main method**
   1. In this class, create a couple of Baker objects, one of which should be:
      * Barbara who specialises in Bakewell Tarts.
   2. Call the display method on each Baker object, it should print a message relating to how the sword attacks (e.g. “Barbara specialises in Bakewell Tarts”).
2. Let’s now create another class that **extends** from TradesPerson but has some members and methods that are different from that of a **Baker**.
   1. **Create** a class named **Bulder** that extends **TradesPerson**, but includes a list of tools. An array of Strings as a member.

String [] tools = {"trowel", "hod", "shovel"};

* 1. Add a constructor that calls the superclass’ constructor, passing “builder” for the trade.
  2. Add a **uses** method that prints out “<builder’s name> uses … “ followed by each tool in the array
  3. **Override** the inherited **display** method so that it calls the **uses** method and displays all information about a builder
  4. Back in your **InheritanceDemo** class, create a builder object in the main method and call its display method – you should see the builder’s details and list of tools.

# Exercise 3 – Shape Inheritance Hierarchy



This exercise should be done in **Processing**. Download shapes.zip from **Moodle**. Run the supplied program and observe the output (Figure 4). You should see stars and circles that move around the canvas in a pleasing manner.

A picture containing chart

Description automatically generated

Figure Shapes.zip Starter Code Output

## Program Overview

This subsection provides an overview of the supplied code before delving into the exercise. If you have read through the code and understand it, you can go straight to the next section.

Currently, the **RandomShapes** program has two classes – Circle and Star – the UML Class diagrams for these classes are depicted in Figure 5**Error! Reference source not found.**.

The **Circle** class contains 6 variables to record the position of each circle, the size, colour, and an xt and yt variable (the xt and yt variables will store the location the object will move to). The class also contains methods to move, render, and update – each of these methods do what you would expect.

The **Star** class contains the same 6 variables and 3 methods, but also contains a variable to record the number of points a star object has.

The main tab contains two arrays – one to store **circles**, the other to store **stars**. Each of these can store 50 of their respective objects (note: the arrays will store references to objects, not the objects themselves). Two loops in the setup procedure deal with creating 50 circle objects and 50 star objects, respectively. The draw procedure renders a white background and then loops through each of the arrays (using a for-each / enhanced for loop) and calls the update method on each object in the array.

Recall from the lecture, we can place common attributes and methods that similar classes have in common and move them into a base class (referred to as a superclass).

Diagram

Description automatically generated with medium confidence

Figure Circle and Shape classes - no inheritance

## Exercise 3.1 – Shape Class

1. Create a new tab named **Shape**. In this tab, create a class named **Shape**
2. Remove the variables that the **Circle** and **Star** class have in common and place these into the **Shape** class – this will serve as the superclass for the Circle and Star classes
3. Add the relevant code to the top of the **Circle** and **Star** class so that they both **extend** the **Shape** class. This will cause both of these classes to inherit any variables and methods in the **Shape** class
4. Every **Circle** and **Star** moves in the same way – move the **move** method to the Shape superclass and remove these definitions from the **Circle** and **Star** classes
5. The update method is also the same for the **Circle** and **Star** classes – place this method in the superclass and remove it from the subclasses
6. You will notice that the **Shape** class does not have a render method, meaning render is underlined red in Processing, since it is not defined in the Shape superclass. Add a **render** method to the **Shape** class that does nothing (has an empty body) – we will see how we can improve this later.
7. Add the constructor below to the **Shape** class. This constructor will be responsible for initialising a shape object’s x, y, size, and colour variables – something that every Shape has

Shape( float x, float y, float size, color colour) {

this.x = x;

this.y = y;

this.size = size;

this.colour = colour;

// set a target to move towards

xt = x;

yt = y;

}

1. We can now modify the constructors in the **Circle** and **Star** classes. Recall form the slides that a class can call the constructor of its immediate superclass via the **super** keyword. In our case, the superclass constructor deals with x, y, size, and colour
   1. **Circle** class – replace the code in the **Circle** constructor with a single line of code that calls the superclass’ (Shape) constructor
   2. **Star** class – replace the code in the **Star** constructor with a line of code that calls the superclass’ constructor – your **Star** constructor should take care of initialising **numPoints** still – since not every Shape has that

At this point, your program should resemble the UML diagram depicted in Figure 6.

Diagram

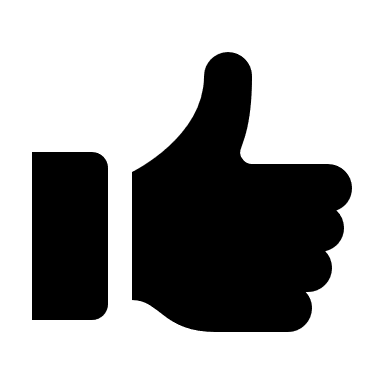
Description automatically generated

Figure RandomShapes Inheritance Design

## Exercise 3.2 – Square Class

We will now create a new class which will inherit from the **Shape** class.

**Good Programming Practice**Although the @Override annotation is optional, it is good practice to use it. The compiler will check that the method you are overriding exists in the superclass to ensure you are overriding an inherited method.



1. Create a new tab named **Square** and class named **Square** that *extends* the **Shape** class.
2. Define a constructor that accepts floats for x, y, size, colour (this constructor should call the superclass constructor).
3. Override the inherited **render** method so that it draws a square (you could either use the built-in square command or rect).
4. Override the inherited **move** method so that the movement of a square differs to that of a **Circle** or **Star** (e.g. you can move the x and y by random values).
5. Create an array of 50 squares in the main tab, create the objects using a for loop within setup, and then use a loop within draw to render the squares.

## Exercise 3.3 – Declaring abstractness

Currently, the **render** method of the **Shape** class has an empty body. Every subclass of the **Shape** class overrides this method – which makes sense since every type of Shape *looks* different – but what exactly does a **Shape** *look* like?

Since the render method of the Shape class does not provide a concrete implementation, we can declare it as **abstract.** The moment you declare any method in a class as being abstract, the entire class must be declared abstract – as the class contains some incomplete implementation.

Replace the empty **render** method in the **Shape** class with the line of code below:

**abstract void** render();

Since abstract methods do not provide implementations, they do not have bodies, meaning we only provide the method header, followed by a semi-colon. It will be up to the subclasses of the Shape class to provide implementation details for this method (which all of our subclasses do).

As mentioned earlier, the moment we declare a method as abstract, we must declare the entire class as abstract - because the class now contains some incomplete implementation. Add the abstract keyword before the class keyword:

**abstract class** Shape

What happens If you try to create an object of the Shape class now?



# Extension Exercise – Overriding an Object’s toString() method

This extension exercise will require you to revisit your program from Exercise 2 (the weapon inheritance hierarchy).

In the main method, add some print statements that print each of the weapons (i.e pass the variable name of the object into a System.out.println). E.g if you have a weapon named **sword1,** your print statement will read **System.out.println(sword1);** You should see output that resembles that of Figure 7 – where the Class name is printed (Sword or Bow), followed by an @ symbol and a code.

Graphical user interface, text, application

Description automatically generated

Figure Result of printing out object references

Every class in Java – whether we are the ones who created it or not – extends from a class called **Object**. This **Object** class sits at the very top of the inheritance hierarchy. Although the direct superclass of our **Sword** and **Bow** classes is **Weapon**, their indirect superclass is the **Object** class. The **Weapon**’s direct superclass is also **Object**.

## The Object Superclass

The object superclass has quite a few methods declared in it, one of which is a method called **toString,** which returns a String representation of an object. Whenever we print a variable to the console that points to an object (i.e. a reference), the object’s **toString** method is implicitly called. The default implementation of this method is to show the object’s class name and its hash code (memory address).

1. Override the **toString** method in the **Weapon** class so that the method returns a String representation of the object (the **name**, **minDamage**, and **maxDamage**). This should be similar to that of the output in Figure 8.
2. Override the **toString** method in the **Sword** class. This should make a call to its superclass’ **toString** method (Weapon), and concatenate (+) the things that a sword has (**hiltLength**).
3. Override the **toString** method in the **Bow** class. This should make a call to its superclass’ **toString** method (**Weapon**), and concatenate (+) the things that a bow has (**wood**, and **bowStringLength**).

Graphical user interface, text, application, email

Description automatically generated

Figure Result of printing objects after overriding toString method