Inheritance with Java

Student Workbook

Version 2.1

Table of Contents

Module 1

The Need for Inheritance

Section 1–1

Inheritance

Why Do We Need Inheritance?

- **Software can be written without objects and without inheritance**
  - Mainframe programming languages did not have the concept of objects
- **In Procedural programming Copy/Paste was king**
  - If you had a piece of logic in your code that you needed somewhere else, you just copied it and pasted where you needed it.
  - What are the potential problems?
    - Many copies of the same code
    - What if the logic needs to be changed?
- **OOP introduced Encapsulation, and with it the concept of requiring objects to have specific responsibilities**
  - This allows functions to be organized in a more centralized way
    - Instead of copying and pasting code, we refer to the object that owns the function to do the work
  - Some object based languages like Visual Basic did not support Inheritance
    - So even without inheritance complex software can be written
  - Organizing functions into classes/objects helped simplify the code, but it did not fully solve the problem
    - What if you have 2 or more objects that need to do the same, or similar things?

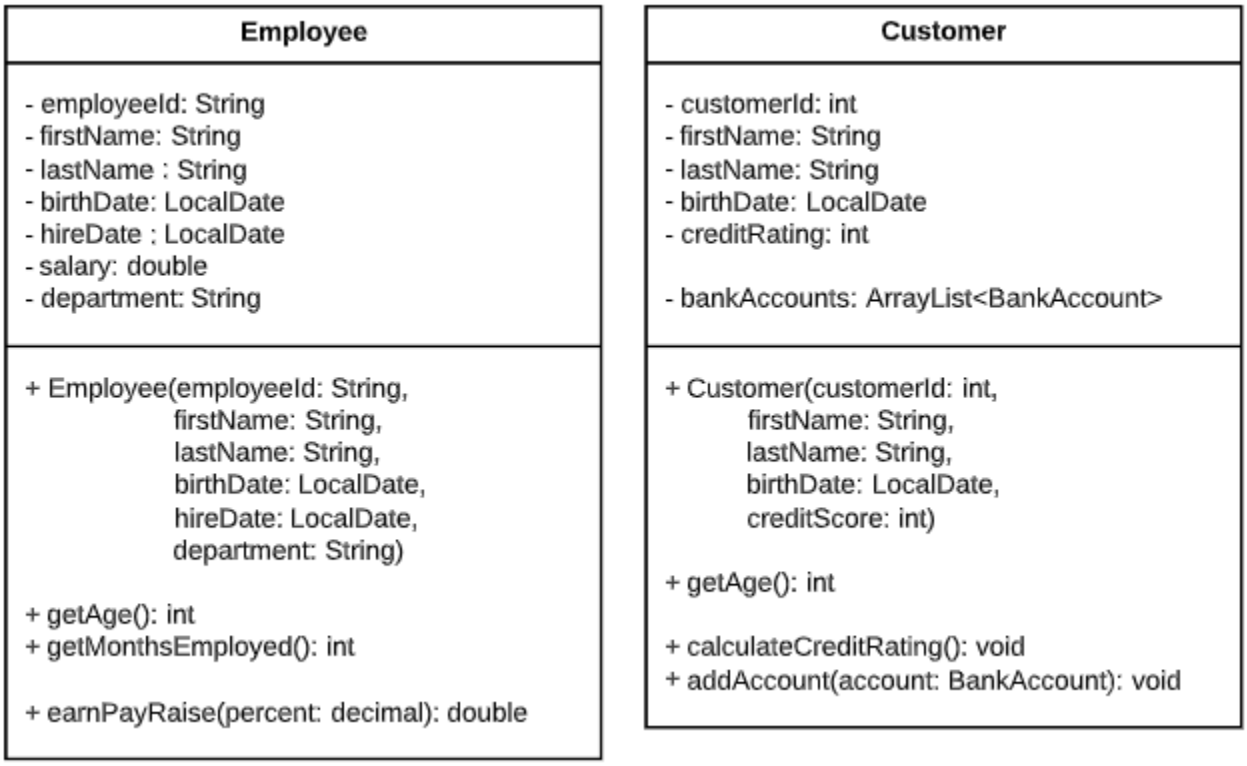
The Need for Inheritance

- **Often we have multiple classes which do similar things**

Example

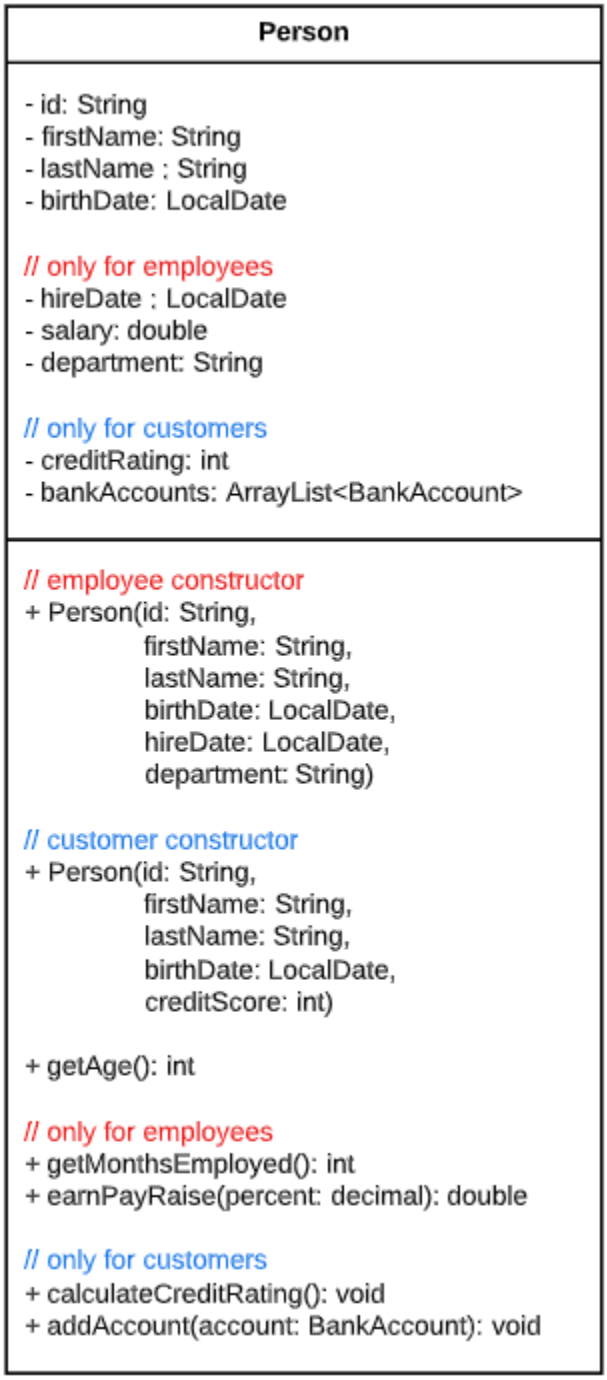
These classes have data and functionality in common, but because they serve different purposes, we have created separate classes.

NOTE: this violates the DRY principle (Don't Repeat Yourself)



The Need for Inheritance *cont'd*

- **One option we have is to create a single class so that we have all shared functionality in on spot**
  - Now there is no duplication of code

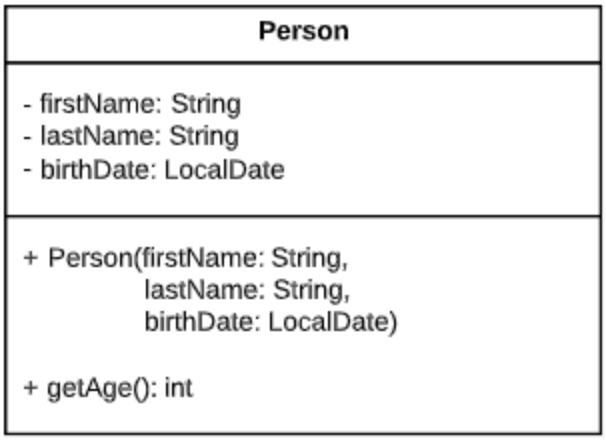


The Need for Inheritance *cont'd*

- **There is a new problem**
  - Our class is no longer cohesive
  - The variables and methods are not all related to each other
- **In OOP we should not sacrifice one principle for the sake of another**
  - DRY vs Cohesion
- **Inheritance allows us to create classes that are both Cohesive and DRY**

Inheritance

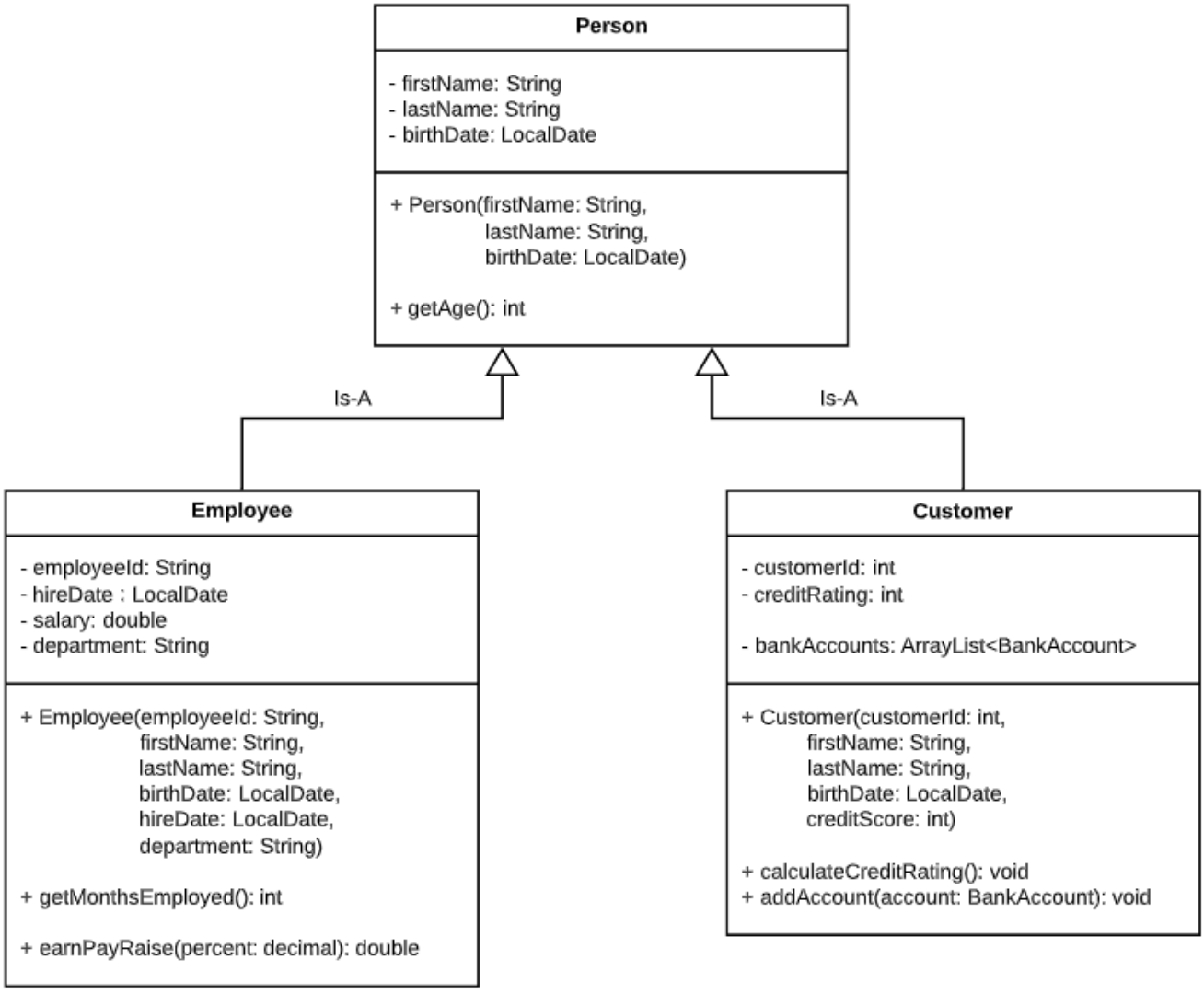
- **Inheritance gives developers the ability to create a super class with all of the shared functionality**
  - Find all of the variables and methods that all classes have in common
  - Create a new class that holds only those variables and methods
  - This process is called generalization because we are creating a "general" class that holds all shared functionality



- **Then create classes which inherit from the Super class**
  - These classes should only add variables and methods that are unique to them
  - What makes an Employee different from a Customer?
  - What makes a Customer different from an Employee?

Inheritance *cont'd*

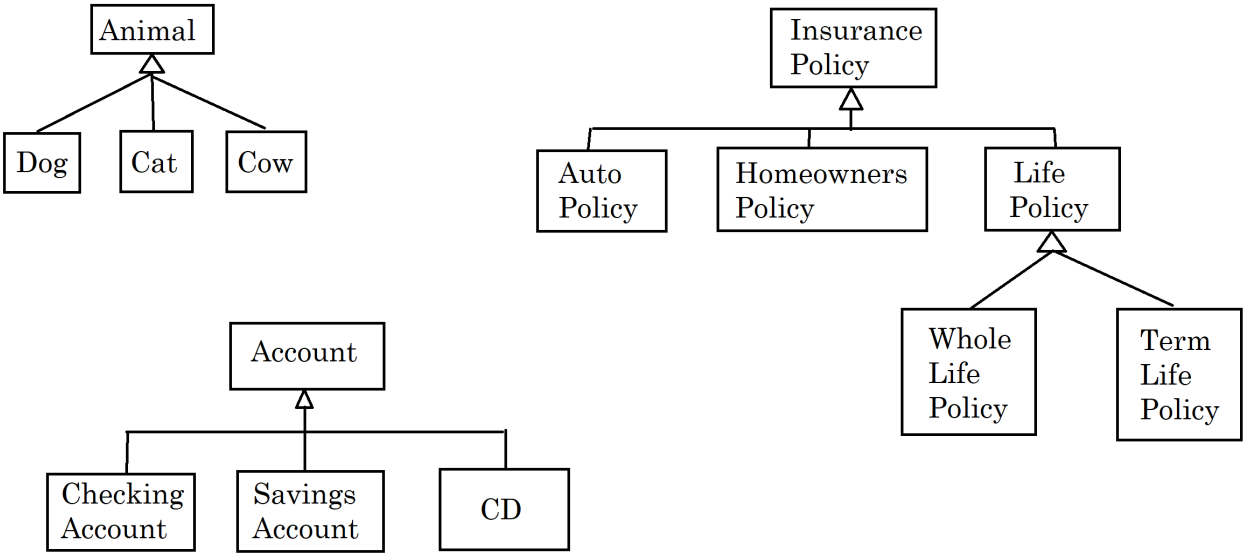
- **Inheritance is called an Is-A relationship.**
  - An `Employee` Is-A `Person`
  - A `Customer` Is-A `Person`



Inheritance Examples

- **The new class *inherits* the attributes and methods from is base, or parent, class**
  - The ***parent class*** is also called the super class or base class and contains the features you want to extend
  - The new class is called the ***child class***, sub-class, or derived class and extends the parent class by adding additional features and overriding existing ones

- **Examples of inheritance include:**



Exercise

Exercise

Work in groups to complete this exercise. When you are finished, come back as a group and each group will share your findings. Diagram your classes for this exercise. You can use notepad or a tool like **diagrams.net** or **lucidchart.com**.

Step 1

Consider that you are creating a game and you need to provide transportation for your characters. Design classes for the following modes of transportation. Take your time to think about what information and functionality is important to each item. Think about how your character will need to interact with the item, and how they will best be able to navigate the game.

Think of each class independent of the others as you go through step 1. Think of this as your rough draft.

- Moped
- Car
- Semi-Truck
- Hovercraft

Step 2

Compare your classes and find the variables and methods that your classes have in common. If only some of the classes have common functionality, that is ok, try to find any duplication in your design. Remember that you can have more than one level of inheritance

Step 3

Create a super/parent class for all variables and methods that ALL of your items have in common. Then create your child classes and have them inherit from the parent. Focus on what makes each class different from the others. If there is any remaining duplication of code, try to see what you can do to extract those methods into a parent class.

Section 1–2

CodeWars

CodeWars Kata

- **Is it Even?**
  - Determine if a given number is even - but there is a twist
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/555a67db74814aa4ee0001b5/java

Module 2

Java Inheritance

Section 2–1

Simple Inheritance

Java Inheritance using `extends`

- **Java uses the `extends` keyword to indicate a class inherits from another class**

  Syntax

```
class SomeSuperClcass {
   ...
}

class SomeChildClass extends SomeSuperClcass {
   ...
}
```

- **All of the features except constructors are inherited into the child class**

  – Child classes must define their own constructors

  Syntax

```
class Asset {
   ...
}

class House extends Asset {
   ...
}

class Stock extends Asset {
   ...
}

class Cow extends Asset {
   ...
}
```

Example: Simple Inheritance

Example

Animal.java

```
public class Animal {
   private String name;

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

Dog.java

```
public class Dog extends Animal {
   private String breed;

   public void setBreed(String breed) {
      this.breed = breed;
   }

   public String getBreed() {
      return breed;
   }

   public void bark() {
      System.out.println("Ruff, ruff!");
   }
}
```

Program.java

```
public class Program {

   public static void main(String[] args) {

      Animal ani = new Animal();
      ani.setName("Bob");

      Dog dog = new Dog();
      dog.setName("Rubby");    // inherited method
      dog.setBreed("Corgi");
      dog.bark();
   }
}
```

`protected` Access Specifier

- **When a parent class member is declared as `private`, child classes inherit it… but they can't access it**

- **Occasionally, a programmer will declare a parent class member with the `protected` access specifier**
  - Child classes can access the member
  - Outside classes can't access the member (unless it is public!)

Example

`Animal.java`

```
public class Animal {
   protected String name;

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

Dog.java

```
public class Dog extends Animal {
   private String breed;

   public void setBreed(String breed) {
      this.breed = breed;
   }

   public String getBreed() {
      return breed;
   }

   public void bark() {
      System.out.println(name + " says ruff, ruff!");
   }
}
```

- **Although this is possible, this isn't done often**

- **Why?**
  - When a class makes a data member protected, it can't control what child classes do to it (i.e. manage the range or value or format of the data)

Exercise

Create a new folder in your `pluralsight` folder, name it `workbook-5`. Do your work in this folder

EXERCISE 1

Part 1

Create a new Java application named Vehicles. You will now create the classes, and appropriate inheritance structure, that you designed in the previous module.

For this exercises none of your classes should have a constructor (we will add those in the next exercise).

The parent class name should be `Vehicle` and you should also create classes for `Moped, Car, SemiTruck` and `Hovercraft`.

If your `Vehicle` class does not have the following properties make sure that you add them (only to the `Vehicle` class). Also add getters and setters for each property you have defined.

```
String color;
int numberOfPassengers;
int cargoCapacity;
int fuelCapacity;
```

Part 2

In your static void main() method create an instance of each of your vehicle types. Using the setters, set some values for the different vehicles. Here are a few examples:

```
Moped slowRide = new Moped();

slowRide.setColor("Red");

slowRide.setFuelCapacity(5);
```

Test the functionality of your other classes and methods.

Why are you able to access the Vehicle getters/setters when you did not create an instance of the Vehicle class?

Section 2–2

Working with Constructors

Constructors and Inheritance

- **You might recall that, in Java, if a class doesn't have a constructor then a "default constructor" is generated under the hood**
  - It has no parameters
  - This is what allows you to instantiate an object

Example

Animal.java

```
public class Animal {
   private String name;

   // there is no explicit constructor in this class

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

elsewhere

```
Animal ani = new Animal();
```

- **Child class constructors automatically call their super class parameterless constructor -- regardless of whether the compiler generated it or the programmer**
  - This happens at the beginning of the execution of the child class construction process

Example

Animal.java

```
public class Animal {
   private String name;

   public Animal() {
      System.out.println("Trace -- in Animal class");
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

Dog.java

```
public class Dog extends Animal {
   private String breed;

   // no constructor

   public void setBreed(String breed) {
      this.breed = breed;
   }

   public String getBreed() {
      return breed;
   }

   public void bark() {
      System.out.println("Ruff, ruff!");
   }
}
```

Program.java

```
public class Program {

   public static void main(String[] args) {

      Animal ani = new Animal();

      Dog dog = new Dog();
   }
}
```

OUTPUT

```
Trace -- in Animal class
Trace -- in Animal class
```

- **When you code a constructor in your child class, you should *explicitly* call the parent class constructor using the function `super()`**
  - It must be the first line in the constructor
  - If you are calling the parent class parameterized constructor, you can pass arguments to it

Example

Dog.java

```
public class Dog extends Animal {
   private String breed;

   public Dog(String name, String breed) {
      super(name);        // calls Animal constructor
      this.breed = breed
   }
   public void setBreed(String breed) {
      this.breed = breed;
   }
}
```

Example: Passing Arguments to Parent Class Constructor

Example

Animal.java

```
public class Animal {
   private String name;

   public Animal() {
      System.out.println("Trace -- in Animal() class");
   }

   public Animal(String name) {
      System.out.println(
         "Trace -- in Animal(name) class w/ name");
      this.name = name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

Dog.java

```
public class Dog extends Animal {
   private String breed;

   public Dog() {
      super();
      System.out.println("Trace -- in Dog() class");
   }

   public Dog(String name, String breed) {
      super(name);
      System.out.println(
         "Trace -- in Dog(name, breed) class w/ name");
      this.breed = breed;
   }

   public void setBreed(String breed) {
      this.breed = breed;
   }

   public String getBreed() {
      return breed;
   }

   public void bark() {
      System.out.println("Ruff, ruff!");
   }
}
```

Program.java

```
public class Program {

   public static void main(String[] args) {

      Animal animal_1 = new Animal();
      System.out.println(
        "animal_1's name is " + animal_1.getName() + "\n");

      Animal animal_2 = new Animal("Elmo");
      System.out.println(
         "animal_2's name is " + animal_2.getName() + "\n");

      Dog dog_1 = new Dog();
      System.out.println("dog_1's name is " + dog_1.getName()
         + " and their breed is " + dog_1.getBreed() + "\n");

      Dog dog_2 = new Dog("Fido", "Mutt");
      System.out.println("dog_2's name is " + dog_2.getName()
         + " and their breed is " + dog_2.getBreed() + "\n");
   }
}
```

OUTPUT

```
Trace -- in Animal() class
animal_1's name is null

Trace -- in Animal(name) class w/ name
animal_2's name is Elmo

Trace -- in Animal() class
Trace -- in Dog() class
dog_1's name is null and their breed is null

Trace -- in Animal(name) class w/ name
Trace -- in Dog(name, breed) class w/ name
dog_2's name is Fido and their breed is Mutt
```

Exercise

EXERCISE 1

Continue working on the Vehicles project from the previous exercise.

Create a constructor in the `Vehicle` class that takes input parameters for each of the private variables in the class.

Create appropriate constructors in each of your inherited child classes and call the constructor super constructor.

Notice that you CANNOT extend the `Vehicle` class unless you explicitly call the super constructor from the child class.

Section 2–3

CodeWars

CodeWars Kata

- **Find the Average**
  - Given an array of integers, find the average of all numbers
  - Return the answer as a double
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/57a2013acf1fa5bfc4000921/java

Module 3

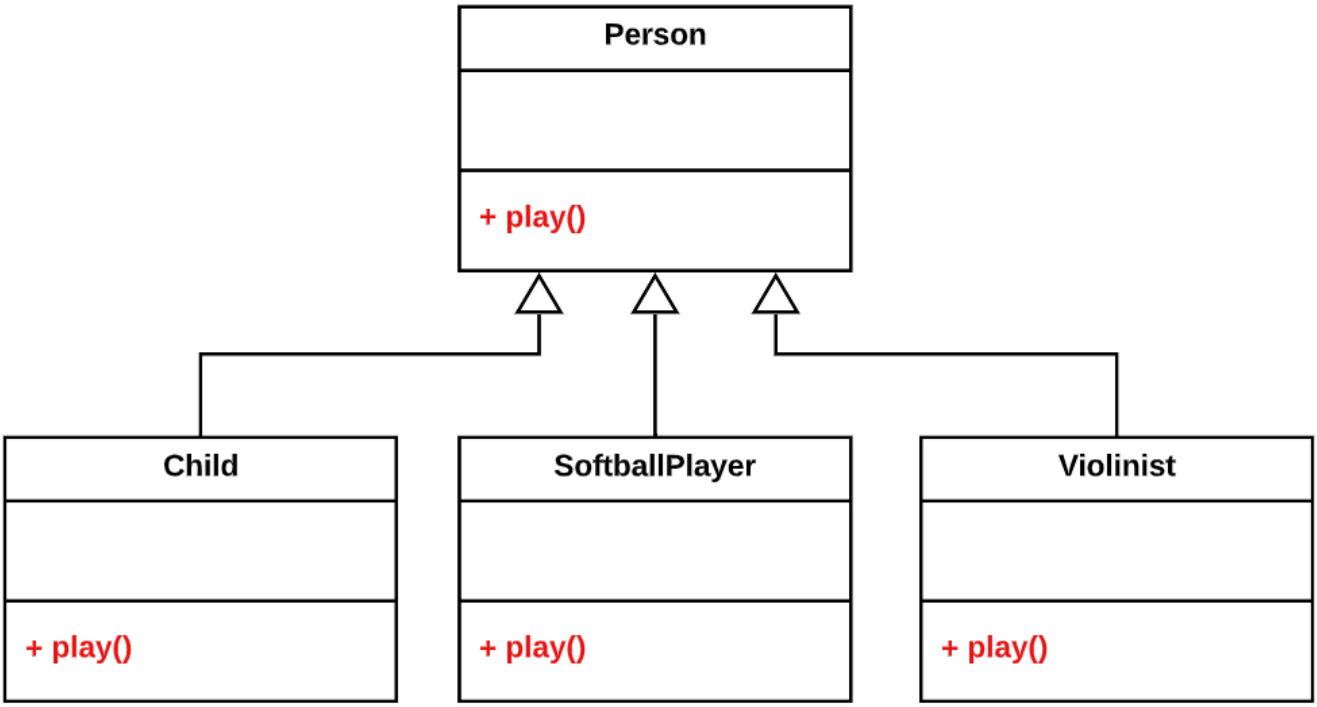
Polymorphism

Section 3–1

Overriding Methods

Introduction to Polymorphism

- **What is Polymorphism?**
  - Polymorphism defines something as having the ability to take on many different forms
- **Remember that inheritance defines an Is-A relationship which points from the child to the parent**
  - This means that all children of the parent implicitly have all of the capabilities of the parent class
    - An Employee Is-A Person
    - A Customer Is-A Person
- **Child classes `extend` the functionality of the parent**
  - This means that child classes have MORE abilities than the parent
  - Child classes also have the ability to CHANGE the way that inherited methods work
- **This ability to change existing functionality is Polymorphism**
  - The Is-A relationship points from child to parent
  - The Polymorphic relationship points from parent to child



- **In the above diagram shows that 3 classes inherit from Person - each class overrides the behavior of the `play()` method**
  - These might be the behaviors of each Person when they **`play`**
    - When a Person is just a Person - they may **run** when they `play()`
    - When a Person is a Child - they may **eat sand**
    - When a Person is a SoftballPlayer - they may **hit a home run**
    - When a Person is a Violinist - they may **play a sonata**

- **How the person behaves is entirely dependent on the context of their current role**

Overriding Inherited Methods

- **Child classes can override inherited methods**
  - Overriding means they can code their own version and it replaces the inherited one

Example

Human.java

```
public class Human {

    public void eat() {
        System.out.println("Me hungry...");
    }
}
```

Caveman.java

```
public class Caveman extends Human {

    public void eat() {
        System.out.println("Me hunt lion and eat!");
    }
}
```

ModernPerson.java

```
public class ModernPerson extends Human {

    public void eat() {
        System.out.println("Me going to McDonalds and eat!");
    }
}
```

Program.java

```
public class Program {

    public static void main(String[] args)() {
        Human h = new Human();
        h.eat();
        Caveman c = new Caveman();
        c.eat();

        ModernPerson m = new ModernPerson();
        m.eat();
    }
}
```

OUTPUT

```
Me hungry...
Me hunt lion and eat!
Me going to McDonalds and eat!
```

Polymorphism and Base Class References

- **The ability to override methods introduces polymorphism into our class library**
  - Polymorphism, from the Greek "many forms", allows us to perform a single action in different ways
  - That is, we can have many forms of the same method
- **It is most notable when we use a base class variable to reference a derived class object**
- **A base class reference variable can refer to any object of a derived type**
  - For example, an animal variable can reference a cat or a dog object because "a cat is-an animal" and "a dog is-an animal"

Example

```
Animal animal;

animal = new Dog();   // okay if Dog extends Animal
animal = new Cat();   // okay if Cat extends Animal
```

- **However, the opposite isn't true**
  - You cannot assign a base class object to a derived class variable
  - What if the assignment isn't valid?

Polymorphism and Base Class References
cont'd

Example

```
Animal animal = /* some value not shown */;

Dog dog = animal;   // ERROR -- what if animal references a Cat
```

- **If you are 100% sure that the base class variable references the right type of object, then you can force the assignment by downcasting**
  - This will cause a runtime error if you are wrong

Example

```
Animal animal = /* some value not shown */;

Dog dog = (Dog) animal;
```

Example: Polymorphism in Action

- **In the example below, you would think that the call `human.eat()` in the feed method would call the `Human`'s `eat()` method**
  - But it doesn't!

- **Since `eat()` is an overridden method, the Java engine decides at runtime which `eat()` method to call based on the *type of* the parameter object**

Example

`Program.java`

```
public class Program {

    public static void main(String[] args)() {
        Human h = new Human();
        feed("Human", h);

        Caveman c = new Caveman();
        feed("Caveman", c);

        ModernPerson m = new ModernPerson();
        feed("Modern Person", m);
    }

    public static void feed(String label, Human human) {
       System.out.print(label + "--> ");
       human.eat();
    }
}
```

`OUTPUT`

```
Human--> Me hungry...
Caveman--> Me hunt lion and eat!
Modern Person--> Me going to McDonalds and eat!
```

Heterogeneous Collections

- **This polymorphic nature of overridden methods leads to some interesting software designs**

- **Now we can hold arrays and other collections using a base class reference**
  - However, the objects in the collection will behave appropriately for their type

Example

Program.java

```
import java.util.ArrayList;

public class Program {

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

        ArrayList<Human> people = new ArrayList<Human>();

        Human h = new Human();
        people.add(h);

        Caveman c = new Caveman();
        people.add(c);

        ModernPerson m = new ModernPerson();
        people.add(m);

        // people is a collection of different types
        // of Human objects

        for(int i = 0; i < people.size(); i++) {
           people.get(i).eat();    // polymorphic behavior
        }
    }
}
```

OUTPUT

```
Me hungry...
Me hunt lion and eat!
Me going to McDonalds and eat!
```

`instanceof` Operator

- **Java provides an `instanceof` operator to compare an object to a specified type**
  - You can use it to see if an object is an instance of a class, a subclass, or even a class that implements an interface (coming soon)
- **In the following example, we will ask each object if they are:**
  - `Human`
  - `Caveman`
  - `ModernPerson`
- **A `ModernPerson` object will answer yes to `ModernPerson` and `Human` because of their inheritance chain**

Example: `instanceof` Operator

Example

Program.java

```
import java.util.ArrayList;

public class Program {

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

        ArrayList<Human> people = new ArrayList<Human>();

        people.add(new Human());
        people.add(new Caveman());
        people.add(new ModernPerson());

        for(int i = 0; i < 3; i ++) {
            if (people.get(i) instanceof Human) {
               System.out.print("Human--> ");
            }
            if (people.get(i) instanceof Caveman) {
                System.out.print("Caveman--> ");
            }
            if (people.get(i) instanceof ModernPerson) {
                System.out.print("ModernPerson--> ");
            }
            people.get(i).eat();
        }
    }
}
```

OUTPUT

```
Human--> Me hungry...
Human--> Caveman--> Me hunt lion and eat!
Human--> ModernPerson--> Me going to McDonalds and eat!
```

More About `instanceof`

- **If you wanted a single answer to an instanceof query, ask about the most specific types first and use and else if structure that would stop questioning once there was a match**

Example

Program.java

```
import java.util.ArrayList;

public class Program {

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

        ArrayList<Human> people = new ArrayList<Human>();

        people.add(new Human());
        people.add(new Caveman());
        people.add(new ModernPerson());

        for(int i = 0; i < 3; i ++) {
            if (people.get(i) instanceof ModernPerson) {
                System.out.print("ModernPerson--> ");
            }
            else if (people.get(i) instanceof Caveman) {
                System.out.print("Caveman--> ");
            }
            else if (people.get(i) instanceof Human) {
               System.out.print("Human--> ");
            }
            people.get(i).eat();
        }
    }
}
```

OUTPUT

```
Human--> Me hungry...
Caveman--> Me hunt lion and eat!
ModernPerson--> Me going to McDonalds and eat!
```

getClass()

- **You can call the `getClass()` method on any object and it will return the class name**
  - If the class is defined in a package, it will return its fully qualified name (ex: "java.util.ArrayList")
  - If the class is not defined in a package, it will return the word "class" followed by the class name (ex: "class Human")

Example

Program.java

```
import java.util.ArrayList;

public class Program {

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

        ArrayList<Human> people = new ArrayList<Human>();

        people.add(new Human());
        people.add(new Caveman());
        people.add(new ModernPerson());

        for(int i = 0; i < 3; i ++) {
            System.out.print(
               people.get(i).getClass() + "--> ");
            people.get(i).eat();
        }
    }
}
```

OUTPUT

```
class Human--> Me hungry...
class Caveman--> Me hunt lion and eat!
class ModernPerson--> Me going to McDonalds and eat!
```

`Object` : The Parent Class of all Classes

- **Why can you call `getClass()` on all objects?**
  - In Java, all classes implicitly inherit from a class called Object
- **This gives all Java classes a common ancestor**

Example

```
String s = "Hello";

ArrayList<Vehicle> myList = new ArrayList<Vehicle>();

ModernPerson me = new ModernPerson();

if (s instanceof Object)           // true
if (myList instanceof Object)      // true
if (me instanceof Object)  // true
```

- **`Object` defines some methods that are inherited by all child classes, including:**
  - `equals()`
  - `getClass()`
  - `toString()`

- **The presence of the `toString()` method is why you can display objects in a print statement**
  - The default `toString()` is automatically called
  - Programmers often override `toString()` for a custom implementation

- **For a complete list, see:**
  - https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Exercises

EXERCISE 1

Create a new Java application named AssetManager.

WARNING: This is an academic example and is a hugely simplified version of anything real.

The base class `Asset` can be defined with the following:

Properties:
- description : String
- dateAcquired : String
- originalCost : double

Methods:
- constructor
- all getters / setters
- getValue() : double // returns original cost

You will build two derived classes: `House` and `Vehicle`.

The `House` class inherits from `Asset` can be defined with the following:

Properties:
- address : String
- condition : int (1 -excellent, 2 -good, 3 -fair, 4 -poor)
- squareFoot : int
- lotSize : int

Methods:
- constructor
- all getters / setters
- getValue() : double (override)

```
// A house's value is determined as
// $180.00 per square foot (excellent)
// $130.00 per square foot (good)
// $90.00 per square foot (fair)
// $80.00 per square foot (poor)
// PLUS  25 cents per square foot of lot size
```

The `Vehicle` class inherits from `Asset` can be defined with the following:

Properties: makeModel : String
year : int
odometer : int

Methods: constructor
all getters / setters
getValue() : double (override)

```
// A car's value is determined as
// 0-3 years old  - 3% reduced value of cost per year
// 4-6 years old  - 6% reduced value of cost per year
// 7-10 years old - 8% reduced value of cost per year
// over 10 years old - $1000.00
// MINUS  reduce final value by 25% if over 100,000 miles
//     unless makeModel contains word Honda or Toyota
```

This program will not have a user interface. In your `main()`, create an `ArrayList` of `Asset` objects.

Load it with your `Assets`. Include at least 2 houses (you have a vacation home!) and at least two vehicles. Use descriptions like "my house" or "Tom's truck" for the assets.

Now, loop thru the `Asset` collection displaying the description of each asset, the date you acquired it, how much you paid for it, and its value.

When that works, go back and include in the display either the address of the house or the year and make/model of the vehicle. You will need to use `instanceof` when you loop through the assets to detect the type of asset it is. Once you know it is a House or Vehicle, you will need to downcast it so that you can call the methods of the specific type.

Example

```
String message = "";

if (myAssets.get(i) instanceof House) {
   House house = (House) myAssets.get(i);
   message = "House at " + house.getAddress();
}
else if (myAssets.get(i) instanceof Vehicle) {
   Vehicle vehcile = (Vehicle) myAssets.get(i);
   message = "Vehicle: " +
      vehicle.getYear() + " " + vehicle.getMakeModel();
}
```

`@Override` Annotation

- **Java 1.5 introduced the `@Override` annotation to tell the compiler that the method with the `@Override` MUST override an existing inherited method**
  - If the method doesn't exist, the compiler generates an error

Example

`Human.java`

```
public class Human {

    public void eat() {
        System.out.println("Me hungry...");
    }
}
```

`Caveman.java`

```
public class Caveman extends Human {

    @Override
    public void eat() {
        System.out.println("Me hunt lion and eat!");
    }
}
```

`ModernPerson.java`

```
public class ModernPerson extends Human {

    @Override                          // generates an error!
    public void dine() {
        System.out.println("Me going to McDonalds and eat!");
    }
}
```

Section 3–2

CodeWars

CodeWars Kata

- **Credit Card Mask**
  - Given a credit card number (as a string) mask all numbers except the last 4 digits
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/5412509bd436bd33920011bc/java

Module 4

Abstraction

Section 4–1

Abstraction

Abstraction

- ***Data abstraction*** **is the process of exposing only essential information about an object and hiding implementation details**
  - In Java, this is achieved using the abstract classes and interfaces
- **An abstract class is a class that gathers attributes and methods together, but cannot be used to instantiate an object**
  - It will always be a base class
  - It defines all of the properties and methods that the child classes have in common
  - But it DOES NOT have enough information or functionality to be fully usable
- **You know you are working with an abstract class when someone asks you "what type"?**
  - I'd like to open an account…
  - I'd like to buy an asset…
  - I'd like to take home a mammal…
  - I'd like to apply for a policy…

Abstract Class

- **An abstract class is defined with the keyword `abstract`**
  - That immediately means it can't be used to instantiate an object
  - But you can create child classes that inherit from it

Example

Asset.java

```
public abstract class Asset {
   protected String description;
   protected int yearAcquired;
   protected double originalCost;

   public Asset(String description, int yearAcquired,
                double originalCost {
      this.description = description;
      this.yearAcquired = yearAcquired;
      this.originalCost = originalCost;
   }

   public String getDescription() {
       return description;
   }

   public int getYearAcquired() {
       return yearAcquired;
   }

   public double getValue() {
       return originalCost;
   }
}
```

Elsewhere

```
// you cannot instantiate an abstract class

Asset myAsset = new Asset("Thingy", 2020, 125.00);  // ERROR
```

- **Typically, you create one or more classes that extend the abstract class**

Example

House.java

```
public class House extends Asset {
   private String address;
   private int squareFeet;
   private int lotSize;

   public House(String description, String address,
        int squareFeet, int lotSize,
        int yearAcquired, double originalCost {

      super(description, yearAcquired, originalCost);

      this.address = address;
      this.squareFeet = squareFeet;
      this.lotSize = lotSize;
   }

   public String getAddress() {
       return address;
   }

   public int getYearAcquired() {
       return squareFeet;
   }

   public int getLotSize() {
       return lotSize;
   }

   public double getValue() {
       return (180 * squareFeet) + (0.25 * lotSize);
   }
}
```

Elsewhere

```
// you can instantiate a class that inherits from an
// abstract class

House myHouse = new House("Ranch House", "402 Stevens",
    2000, 43560, 2020, 220000);
```

- **You can also use variables defined as an abstract class type to reference objects from its derived types**
  - This is because of the Is-A relationship defined through inheritance
  - We are guaranteed that every object in the `Asset[]` array `Is-An Asset`
    - So we know that all of the `Asset` behaviors are available to us

Example

```
// Assume House and Jewelry extend the Asset class

Asset[] myAssets = new Asset[3];

myAssets[0] = new House("Ranch House", "402 Stevens",
    2000, 43560, 2020, 125.00);

myAssets[1] = new House("Rental", "3329 Duchess",
    1600, 5445, 1995, 53000);

myAssets[2] = new Jewelry("Ring", "Diamond", 1.5, 1979, 1200);

// Loop thru the assets and get the value of each

double myNetWorth = 0;
for(int i = 0; i < myAssets.length; i++) {
   myNetWorth += myAssets[i].getValue();
}
```

Abstract Methods

- **Abstract methods are methods in an abstract class that are placed there so that all derived type have to provide an implementation of the method**

Example

Asset.java

```
public abstract class Asset {
   protected String description;
   protected int yearAcquired;
   protected double originalCost;

   public Asset(String description, int yearAcquired,
                double originalCost {
      this.description = description;
      this.yearAcquired = yearAcquired;
      this.originalCost = originalCost;
   }

   public String getDescription() {
       return description;
   }

   public int getYearAcquired() {
       return yearAcquired;
   }

   // now child classes *MUST* override this method
   public abstract double getValue();
}
```

Example

House.java

```
public class House extends Asset {
   private String address;
   private int squareFeet;
   private int lotSize;

   public House(String description, String address,
        int squareFeet, int lotSize,
        int yearAcquired, double originalCost {

      super(description, yearAcquired, originalCost);

      this.address = address;
      this.squareFeet = squareFeet;
      this.lotSize = lotSize;
   }

   public String getAddress() {
       return address;
   }

   public int getYearAcquired() {
       return squareFeet;
   }

   public int getLotSize() {
       return lotSize;
   }

   public double getValue() {
       return (180 * squareFeet) + (0.25 * lotSize);
   }
}
```

Leveraging AI Tools for Advanced Java Concepts

- **Now that you're diving into more sophisticated Java concepts like inheritance, abstraction, interfaces, and polymorphism, let's revisit how to effectively use AI tools to enhance your learning experience.**

- **Advancing Your Understanding with AI**

  - As concepts become more complex, AI tools can help you visualize and grasp abstract ideas

  - Remember that understanding these concepts deeply is crucial for your development as a programmer

  - AI can help clarify, but you still need to implement and practice these concepts yourself

- **Effective Strategies for Advanced Topics**
  - **Request visual explanations of relationships**
    - "Can you explain the relationship between abstract classes and interfaces using a diagram?"
    - Use the explanation to create your own diagrams and mental models
  - **Ask for real-world analogies for abstract concepts**
    - "How would you explain polymorphism using a real-world example from sports?"
    - This helps bridge theoretical knowledge with practical understanding
  - **Explore design decisions**
    - "When would I choose to use an interface instead of an abstract class in Java?"
    - Understanding the "why" behind patterns is as important as knowing the syntax

- **Sample Questions for Advanced Topics**
  - **For inheritance hierarchies:**
    - "What are common mistakes when designing inheritance relationships?"
    - "How might I refactor this inheritance structure to be more flexible?" (show your current design)
  - **For polymorphism:**
    - "Can you explain how method overriding enables polymorphism in Java?"
    - "What are the benefits of using polymorphic collections in Java?"
  - **For design patterns:**
    - "How does the Singleton pattern relate to static methods we've learned about?"
    - "What problem does the Strategy pattern solve in object-oriented design?"

- **Example:**

I understand the basic syntax differences between abstract classes and interfaces in Java, but I'm struggling to know when to use each one. Can you explain the decision-making process with examples?"

Reasoned about abstract classes vs interfaces for 13 seconds >

When deciding between abstract classes and interfaces, the choice depends largely on the relationship among your classes and how you want to structure your code. Here's a breakdown of the decision-making process:

Use Abstract Classes When:

- **Shared State and Implementation:**
  Use an abstract class when multiple classes share common fields (state) and behavior. Abstract classes allow you to provide a base implementation for some methods while forcing subclasses to override others.
- **Tightly Related Classes:**
  When your classes have an "is-a" relationship (for example, Dog is an Animal), an abstract class lets you capture shared properties and behaviors.
  Example:

java Copy

```java
public abstract class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }

    public abstract void makeSound();
}

public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}
```

- **Single Inheritance Requirement:**
  Since Java only allows single inheritance for classes, use an abstract class when you want to enforce a specific class hierarchy.

Use Interfaces When:

- **Defining a Contract:**
  Use an interface to define a set of methods that can be implemented by any class, regardless of where it sits in the class hierarchy. This is ideal for defining capabilities or behaviors that might be shared across unrelated classes.
- **Multiple Inheritance of Type:**
  A class in Java can implement multiple interfaces. This is useful when you want a class to adhere to several contracts.
  Example:

java Copy

```java
public interface Flyable {
    void fly();
}

public class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Flying high!");
    }
}

public class Airplane implements Flyable {
    @Override
    public void fly() {
        System.out.println("Taking off!");
    }
}
```

- **Decoupling the Design:**
  Interfaces help you decouple your code. By programming to an interface, you allow for more flexible and modular designs, making it easier to swap implementations without changing the client code.

- # Example continued:

Key Considerations:

- **Commonality vs. Capability:**
  - **Abstract Class:** Use when you have a set of classes that share a common base implementation and state.
  - **Interface:** Use when you want to define a common capability (e.g., "Flyable", "Drivable") that can be applied across different, possibly unrelated classes.
- **Code Reusability:**
  Abstract classes let you reuse code through inheritance. If several classes share the same implementation details, an abstract class is a good choice.
- **Flexibility and Multiple Inheritance:**
  Interfaces allow more flexibility since a single class can implement multiple interfaces, supporting a mix of behaviors.
- **Evolution of Java (Java 8 and later):**
  Note that interfaces can now have default methods (with a provided implementation). However, this does not replace abstract classes; it just gives you a bit more flexibility when adding new methods to interfaces without breaking existing implementations.

Conclusion:

- **Choose an Abstract Class:**
  When you want to create a common base with shared fields and implementations for closely related classes (e.g., different types of animals that share basic behaviors).
- **Choose an Interface:**
  When you need to specify a contract for classes that might come from diverse parts of your application, or when you need to simulate multiple inheritance by allowing classes to implement multiple behaviors (e.g., a class that needs to be both "Serializable" and "Comparable").

By considering the degree of shared behavior, the need for multiple inheritance, and the design goals of your application, you can make an informed choice between abstract classes and interfaces in Java.

Does this help clarify the decision-making process with examples?

- **Remember**

  - Use AI to explore different perspectives on complex topics
  - Always implement concepts yourself to solidify understanding
  - When studying design choices, consider tradeoffs and ask about alternatives
  - Focus on understanding principles rather than memorizing specific implementations
  - Share your code and thinking with AI to get feedback on your approach, not for solutions
  - As you continue advancing in this course, these tools can help you build a deeper understanding of Java's object-oriented features and prepare you for real-world software development challenges!

Exercises

EXERCISE 1

Create a copy of the Assets project and name it AbstractAssets. You will work in the AbstractAssets project.

Part 1

Modify your `Asset` class to be an abstract class. Then make the `getValue()` method an abstract method.

Re-test your application.

Try to create an instance of an `Asset`. What happens?

Part 2

Create a new class that extends `Asset`, the `Cash` class. This represents the cash that you stash away under your mattresses.

The value of cash does not fluctuate, therefore there does not need to be any custom logic for the `getValue()` method.

What issues are you running into when you try to create the `Cash` class? What are your options to address these issues?

Section 4–2

CodeWars

CodeWars Kata

- **Cats and Shelves**
  - Find the least number of times a cat has to jump to make it to the top shelf
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/62c93765cef6f10030dfa92b/java