Inheritance, Abstraction, and Run-time Polymorphism

CS 2110

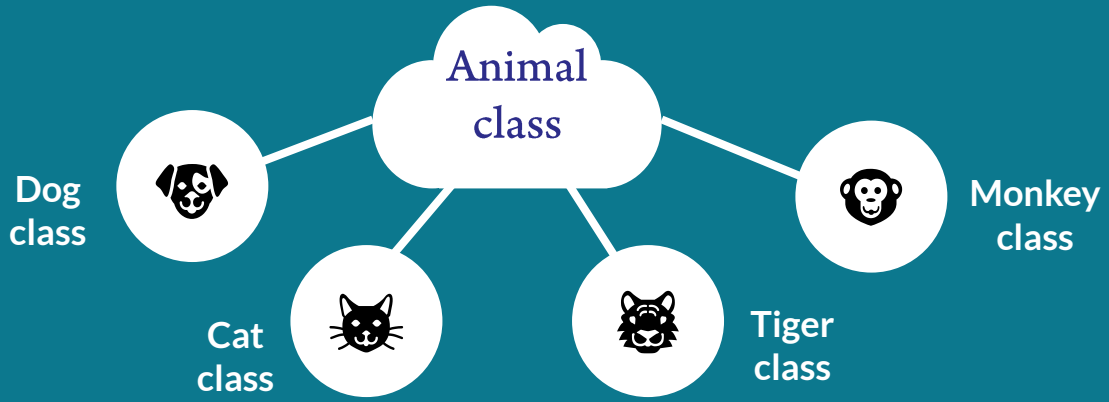
Announcements

- **Homework 2B**:
  - Due tomorrow night at 11:30pm
  - Be sure to review your Homework 1B style/comments feedback to fix for homework 2B
- **Exam Scores**:
  - The **Coding portion** of your score is available (see announcement & check under Assignments)
    - Regrades (coding portion) must be made in instructor office hours
    - If my office hours do not suit you, please contact me and we'll arrange a time to meet that's mutually agreeable
  - The total score will be up later today

Inheritance

Inheritance is an **object-oriented** concept that supports cohesion - code reuse and polymorphic behavior

[Recall!] Inheritance: is-a relationship



- A *subclass* extends a *superclass* (*abstracting common states and behaviors*)
- Use the *is-a test* to verify that your inheritance hierarchy is valid;
  if X extends Y then X is-a Y must make sense
- The *is-a relationship* works only in one direction; a lion *is-a* animal but not all animals are lions

Inheritance Vocabulary

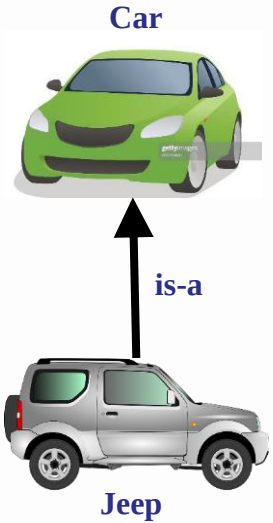
- When a new class is defined from an existing class
  - The new class is called the **subclass** (**derived class** or *child* class)
  - The existing class is called the **superclass** (**base class** or *parent* class)
- We would say the following:
  - The subclass inherits from the superclass.
  - The subclass extends the superclass.
- A note on access modifier: protected
  - A subclass **cannot** access **private** fields or methods of the superclass
  - **Superclass** can allow subclass access by declaring fields/methods as protected (visibility: class itself, all subclasses, within same package)

Substitutability Principle

- We say: any subclass object (e.g., Jeep) **is-a** instance of a superclass object (e.g., Car), and **inherits** its states and behaviors

- Wherever we see a reference to a Car (superclass) object in our code, we can legally replace that with a reference to Jeep (any subclass object)

- Implies that we can **substitute** the subclass object in any way that's legal for the superclass

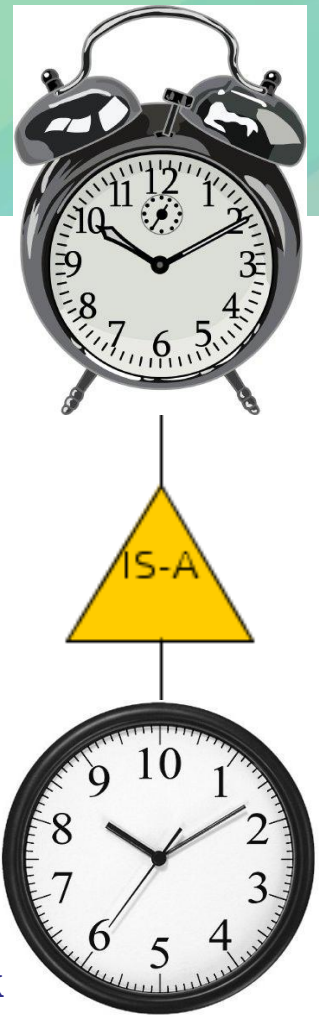


"Is-a" and Inheritance

- We say: any subclass object (AlarmClock etc.) "**is-a**" instance of a superclass object (e.g. Clock), and **inherits** its states and behaviors

- **Substitutability principle**: If AlarmClock is a subtype of Clock, then objects of type Clock may be replaced (substituted) with any object of subtype AlarmClock
- Implies that we can "use" the subclass object in any way that's legal for the superclass

Alarm Clock EXTENDS (is a subclass of) the Clock class

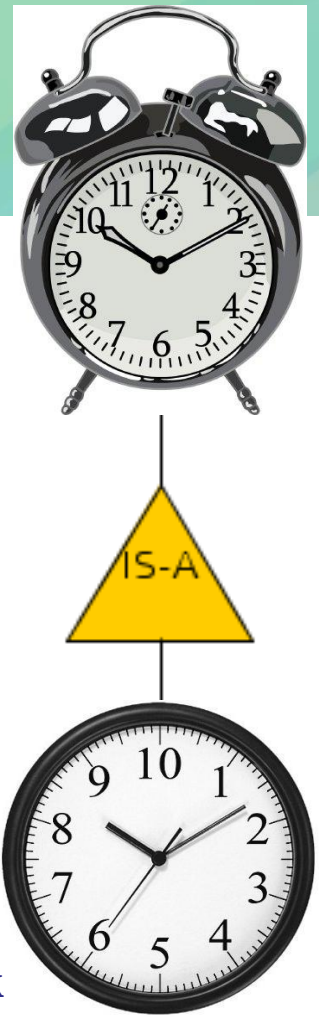


Substitutability principle (example)

```
public void changeTime( Clock c ) {
        ... // some implementation using Clocks
}
```

- If a method takes in an object of type Clock, and if AlarmClock extends Clock, then that method can take an object of type AlarmClock!
- But a function that takes in an AlarmClock cannot take in a non-alarm Clock.

Alarm Clock EXTENDS (is a subclass of) the Clock class



Motivations for Inheritance

- Many times we need a class that is only slightly different from an existing class
  - Don't repeat yourself (DRY)! ~ Write once!
  - Sometimes we just need to add something to the state or add/change the behavior of a method

    → Use inheritance!

Motivations for Inheritance

- **Benefits**: Inheritance can help with the following:
  1. **Code reuse**
     - Our new (subclass/child) class "`extends`" the existing (subclass/parent) class and allows us to **re-use code** that they have in common
  2. **SW that better matches the real world problem**
  3. **Flexible Design**
     - Gives us **flexibility** at run-time in calling operations on objects that might have different types (→run-time polymorphism)

Inheritance: super

How to access a superclass's states and methods

- The subclass object inherits state and behavior from the superclass object, but can **override** these properties
- A subclass object may choose to access the superclass's implementation of its overridden method by using the keyword **super**

```
class Animal {                          // Animal: superclass
      public String getName() {
            return this.name;
}     }
class Cat extends Animal {              // Cat: subclass
      public String getName() {
            return "Meow " + super.getName();
}     }
```

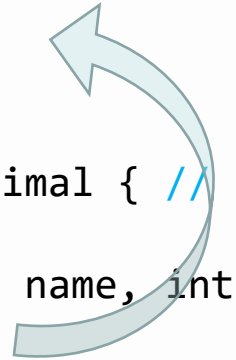
calling

Inheritance: super()

*How to call the superclass's **constructor** method(s)*

- Unless specified otherwise, the subclass constructor calls the superclass constructor **with no arguments** e.g. `super();`
- To call a superclass constructor, use `super()` reserved word as a method. Has to be the first statement of the subclass constructor (can also pass arguments)

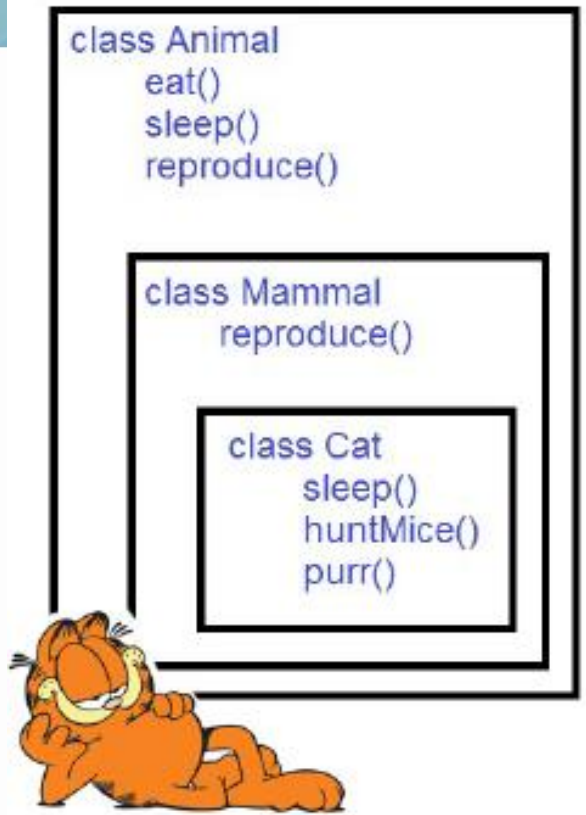
```
class Animal {   // superclass
   String name;
   public Animal(String name) {
      this.name = name;
   }
}

class Cat extends Animal { // subclass
   int hoursOfSleep;
   public Cat(String name, int hrs) {
      super(name);
      this.hoursOfSleep = hrs;
   }
}
```



Implicit super constructor Animal() is undefined. Must explicitly call another constructor

Inheritance code examples: Person.java/Employee.java/GoodEmployee.java; Clock.java and AlarmClock.java

Inheritance: *Animal Example*



Cat garfield = new Cat
(20hrs, NoWay, Always);

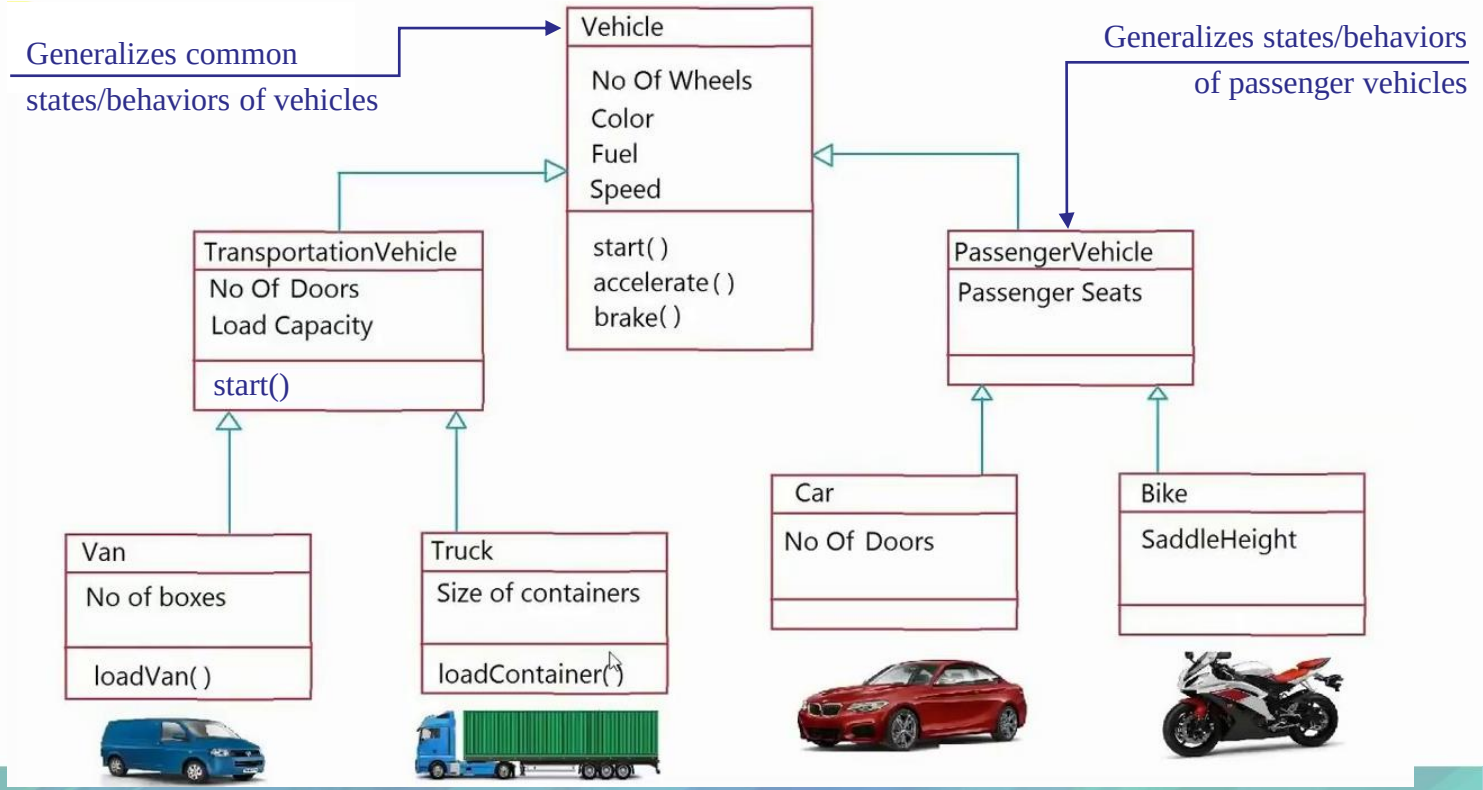
Cat garfield has:
- eat() -- Animal
- reproduce() -- Mammal
- sleep() -- Cat
- huntMice() -- Cat
- purr() -- Cat

Cat objects **inherit** all characteristics of Mammal objects and, in turn, Animal objects.

- sleep() in Cat *overrides* sleep() in Animal (to include long hours and naps)
- reproduce() in Mammal *overrides* reproduce() in Animal (mammals give live birth)

- We can **define** one class in terms of another
- Subclass gets (**inherits**) the state (*fields*) and behavior (*methods*) of the superclass
- We can **add** additional information (fields or methods) to the subclass
- We have the ability to **override** methods in the subclass *to better suit the required functionality* of that class

Abstraction: Generalizing object behavior



Abstract Class

- An abstract class is a class that cannot be directly instantiated
  - It will still have a constructor
- Acts as a means to **encapsulate data** shared among subclasses
- It can serve as a parent class (super class) to another class, meaning it can have:
  - Fields and Working methods
- An abstract class can also have abstract methods
  - An abstract method is unimplemented (a method "stub")
  - Every child class is required to implement it
    - i.e. to be useful a subclass must extend the abstract class
- Code examples: Triangle.java (abstract class); RightTriangle.java and EquilateralTriangle.java (sub classes)

Run-Time Polymorphism

Run-time Polymorphism

- What is **run-time polymorphism**?
  - It is the different effects of invoking the **same method** on **different types** of objects
    - Java asks "**who are you?**" ("what is your data type?")
    - At **run-time**, Java calls the **appropriate** method
    - Could be many methods of the same name in different classes
  - Java provides this through *inheritance* and *interfaces*
    - (Also, related to ***substitutability principle***)

Run-time Polymorphism

Many forms: adapting behavior during run-time

Light-morph jaguar

Dark-morph or melanistic

Calling the getColor() method on two Jaguar objects

```
class LJaguar extends Jaguar {
     public String getColor() {
          return "I'm the light-morph one!";    }
}

class BJaguar extends Jaguar {
     public String getColor() {
          return "I'm the melanistic one!";   }
}

Jaguar j1 = new LJaguar();
Jaguar j2 = new BJaguar();
System.out.println(j2.getColor())
```

Who are you?

Where have we seen this polymorphic behavior before? → Writing the equals() method!

```
public boolean equals (Object other) {
     if (other == null)
          return false; // other is null
     if (other instanceof Dog) {
        Dog d2 = (Dog) other; // cast other to Dog
        if (this.name.equals(d2.name) &&
           this.breed.equals(d2.breed) &&
           this.age == d2.age) {
               return true; // two Dogs are equal
        }
        else
               return false; // two Dogs are not equal
     }
     else {   // other is not a Dog, so return false
        return false;
     }
}
```

.equals()

Is this a recursive call?

Calling equals() method

- Each type has its own associated equals() method

- Yes! You can call .equals() inside the .equals() method you are writing!
- You are **not** calling yourself (not recursion!) you are calling the .equals method of the type associated with the objects you are comparing
  - E.g. `this.name.equals(c2.name); // String equals method`

Polymorphism

name is of type String
(so calling String equals method)

Another Example – What's the output of the code?

- **Run-time polymorphism example:**

```
ArrayList<Object> myList = new ArrayList<Object>();
myList.add(1);                          // add int
myList.add("hello");                    // add String
myList.add(new Object());               // add Object
int i = (Integer) myList.get(0);
String s = (String) myList.get(1);
Object o = myList.get(2);
System.out.println(i + " " + s + " " + o);
```

- Prints:

```
1 hello java.lang.Object@10b62c9
```

Calling appropriate toString() methods for "i", "s", and "o".

Another Run-time Polymorphism Example:

```
public class Animal {
    public void move() {
        S.O.P("Animals can move!");
    }
}

public class Cat extends Animal {
    public void move() {
        S.O.P("Cats can walk & run");
    }
}
```

```
public class TestCat {
    public static void main(String
                                args[]) {
        //Animal reference & object
        Animal a = new Animal();
        //Animal reference, Cat obj.
        Animal b = new Cat();

        a.move();// method in Animal
        b.move();// method in Cat
```

OUTPUT: Animals can move!
Cats can walk & run

Benefits of Polymorphism

Why Polymorphism is good in Software Engineering

- **Simplicity**
  - When writing code that deals with a family of types, the code can ignore type-specific details and just interact with the family base type
  - Code becomes easier to write and easier for others to understand
- **Extensibility**
  - Other subclasses can be added later to the family of types, and objects of those new subclasses would work with the existing code
  - Code becomes easier for you and others to extend its functionality

Polymorphism Types

Type	Description
Method *overriding*	Methods of a subclass override the methods of a superclass
Method overriding of *abstract* methods	Methods of a subclass implement the abstract methods of an abstract class
Method overriding through Java *interfaces*	Methods of a concrete class implement the methods of an *interface*

Coming up: we'll be talking about interfaces

In-Class Activity:
Run-time Polymorphism

In-Class Activity

- Download the .zip archive and install all .java files (Eclipse)
- Read and understand the four classes provided
- Finish the `giveShot()` method with the requirements:
  - Take an animal as an argument
  - Print the animals voice as an output
- Complete the necessary lines in `main()` to print the following output when the **Vet.java** file is executed:

```
Dog Jack after the shot cried Woof!
Cat Garfield after the shot cried Miaw!
Animal Marlow after the shot cried ...
```

- Submit the completed **Vet.java** file