# Object Orientated Programming Details

COMP360

*“The ancestor of every action is a thought.”*

Ralph Waldo Emerson

# Three Pillars of OO Programming

- Inheritance
- Encapsulation
- Polymorphism

# Inheritance

- Inheritance allows a program to build upon existing classes and methods
- Child classes inherit or extend the methods and class data values of the parent class
- If `kitten` inherits from `cat`, then a `kitten` object is also a `cat` object
- Some languages require the programmer to have access to the source code of the parent class (i.e. C++) while others do not (i.e. Java)

# Encapsulation and Data Abstraction

- Classes are described by their interface, that is, what they can do and how they are used
- The internal data and operation are hidden
- In this way if the internal operation is changed, programs using the class will not be impacted as long as the interface remains consistent

# Accessing public Instance Values

```
public class Car {
        double miles, gallons;
        public double mileage;
        // other data and method are not shown
}

// in another program
Car junker = new Car();
double mpg = junker.mileage;
```

# Accessing Instance Values by get Method

```
public class Car {
        double miles, gallons;
        private double mileage;
        public double getMileage() {
                return mileage;
        }
        // other data and method are not shown
}
// in another program
Car junker = new Car();
double mpg = junker.getMileage();
```

# Accessing Instance Values by a Method

```
public class Car {
        double miles, gallons;

        public double getMileage() {
                return miles / gallons;
        }
        // other data and method are not shown
}
// in another program
Car junker = new Car();
double mpg = junker.getMileage();
```

# Polymorphism

- Polymorphic methods can be applied to arguments of different types, but behave differently depending on the type of the argument
- You can have multiple methods with the same name, but different number or type of parameters

# Java Constructors

- The constructor method is called when an object is first created
- Constructors are called by other classes after the keyword new
- If a class does not have a constructor, Java assumes a constructor with no arguments that does nothing

# Polymorphic Constructors

- A class may have multiple constructors with different number or type of arguments

```
public Goat {
        public Goat( ) { … }
        public Goat(String  cat) { … }
        public Goat(double  dog) { … }
}
```

# Removing the default

- If you create a constructor with parameters, then the default no parameter constructor does not work
- You must create a default no parameter constructor if you want one with other constructors

# Missing Default Constructor

```
public class Aardvark {
    public Aardvark(int ant) { … }
}
```

- in another class

```
        // This does not work
Aardvark termite = new Aardvark();
```

# Calling One Constructor from Another

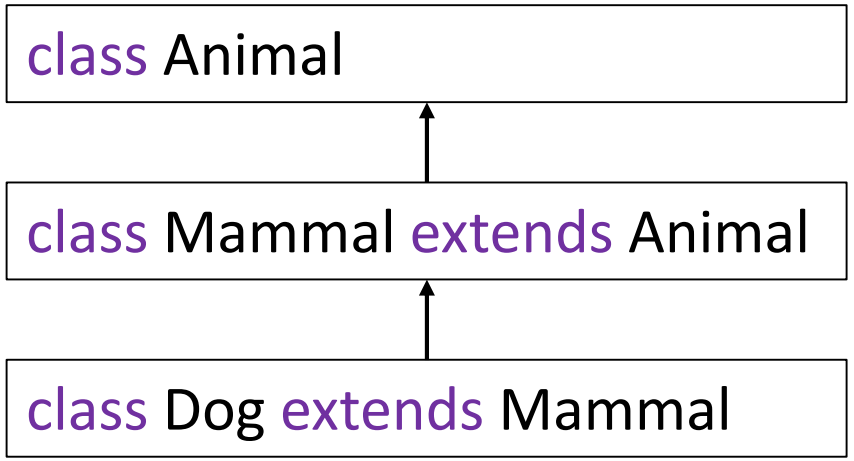
- One constructor can call another constructor of the same class using the method `this( … )`
- When used as a method name, `this` is the constructor for this class

# Constructors calling Constructors

```
public class Vulture {
    public Vulture() {
        this("dead");
    }
    public Vulture(String possum) {
      …
    }
}
```

# Calling Constructor of all Ancestors

- Constructing an instance of a class invokes all the superclasses' constructors up the inheritance chain
- This is called *constructor chaining*




If you create an object of the Dog class, Java will call the constructors of Animal , Mammal and Dog in that order

# Constructors are not Inherited

- Unlike all other methods, constructors are not inherited by subclasses
- If a class does not have a constructor, the constructor of the super classes will be called in order

# What is displayed?

```
class Crow {
    public Crow() {
        this("Crow ");
        System.out.print("caw ");
    }
    public Crow(String starling) {
        System.out.print(starling);
    }
}
class Raven extends Crow {
    public Raven() {
        System.out.print("Raven ");
    }
}
public class ConOrder {
    public static void main(String[] x) {
        Raven bird = new Raven();
    }
}
```

A. Raven
B. caw Raven
C. caw Crow Raven
D. Crow caw Raven
E. Raven Crow caw

# Using the Keyword `super`

- The keyword `super` refers to the superclass of the class in which `super` appears
- This keyword can be used in two ways:
  - To call a superclass constructor
  - To call a superclass method

# Superclass Constructor is super

- You must use the keyword super to call the superclass constructor
- Invoking a superclass constructor's name in a subclass causes a syntax error
- Java requires that the statement that uses the keyword super appear first in the constructor

# super must be first

```
public class Fish {
     public Fish(String dog) {
          System.out.println( dog );
     }
}
public class Trout {
     public Trout( String cat ) {
          System.out.println("trout");
          super( cat ); // Incorrect
     }
}
```

# super must be first

```
public class Fish {
     public Fish(String dog) {
          System.out.println( dog );
     }
}
public class Trout {
     public Trout( String cat ) {
          super( cat ); // correct
          System.out.println("trout");
     }
}
```

# Implicit Calls to Parent Constructors

- If a constructor does not call the super class constructor, Java will automatically call it

```
public class Frog extends Amphibian {
    public Frog() {
      System.out.println("frog");
    }
}
```

- Is the same as

```
public class Frog extends Amphibian {
    public Frog() {
      super( );
      System.out.println("frog");
    }
}
```

# Implicit Constructor Call

```
public class ImplicitCon {
    public static void main(String[] x) {
        Child kid = new Child();
    }
}
class Parent {
    public Parent() {
        System.out.println("Parent");
    }
}
class Child extends Parent {
    public Child() {
        System.out.println("Child");
    }
}
```

Displays

```
Parent
child
```

# Implicit Constructor Call

```
public class ImplicitCon {
    public static void main(String[] x) {
        Child kid = new Child();
    }
}
class Parent {
    public Parent() {
        System.out.println("Parent");
    }
}
class Child extends Parent {

      // no Child constructor

}
```

Displays

**Parent**

# Will this work?

```
public class Apple extends Fruit {
}
class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit"+name);
  }
}
```

A. It should work fine
B. Apple needs a no argument constructor
C. Fruit needs a no argument constructor
D. Both Apple and Fruit need a no argument constructor

# Constructor Chaining

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

# Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

# Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

# Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee's no-arg constructor

# Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

# Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

# Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

# Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

# Trace Execution

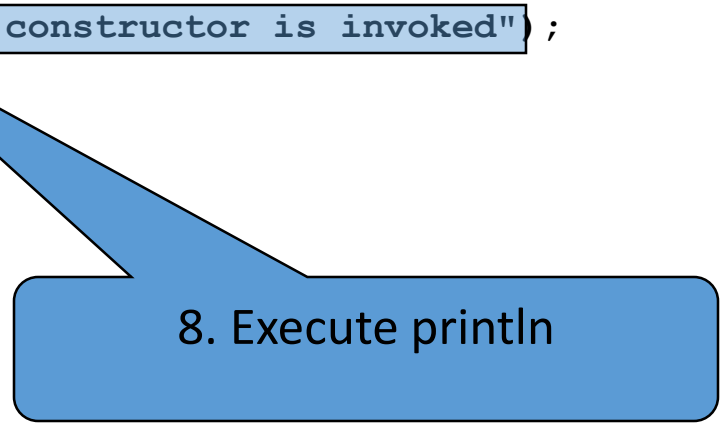
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```



# Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

# Overriding Methods in the Superclass

- A subclass inherits methods from a superclass
- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass
- This is referred to as *method overriding*

# Private Restriction

- A method can be overridden only if it is accessible
- Thus a private method cannot be overridden, because it is not accessible outside its own class
- If a method defined in a subclass is private in its superclass, the two methods are completely unrelated

# Override

- A method is overridden when a child class creates a method of the same name and the same number and type of parameters
- The method of the child class will be used

# Overload

- A method is overloaded when a child class creates a method of the same name but different type or number of parameters
- When a program calls the method, Java will select the method with the correct parameter type

# Override

```
public class OverRide {
    public static void main(String[] x) {
        Cat kitten = new Cat();
        kitten.dog( 3 );
        kitten.dog( 5.0 );
    }
}
class Mammal {
    public void dog( double goat ) {
        System.out.println("Mammal "+goat);
    }
}
class Cat extends Mammal {
    public void dog( double cow ) {
        System.out.println("Cat "+cow);
    }
}
```

# Overload

```
public class OverLoad {
    public static void main(String[] x) {
        Cat kitten = new Cat();
        kitten.dog( 3 );
        kitten.dog( 5.0 );
    }
}
class Mammal {
    public void dog( double goat ) {
        System.out.println("Mammal "+goat);
    }
}
class Cat extends Mammal {
    public void dog( int cow ) {
        System.out.println("Cat "+cow);
    }
}
```

# What is displayed?

```
public class OverRide {
    public static void main(String[] x) {
        Cat kitten = new Cat();
        kitten.dog( 3 );
        kitten.dog( 5.0 );
    }
}
class Mammal {
    public void dog( double goat ) {
        System.out.print("Mammal "+goat);
    }
}
class Cat extends Mammal {
    public void dog( double cow ) {
        System.out.print("Cat "+cow);
    }
}
```

A. Mammal 5.0 Cat 3
B. Cat 3  Mammal 5.0
C. Cat 3.0  Cat 5.0
D. Mammal 3.0  Cat 5.0

# What is displayed?

```
public class OverLoad {
    public static void main(String[] x) {
        Cat kitten = new Cat();
        kitten.dog( 3 );
        kitten.dog( 5.0 );
    }
}
class Mammal {
    public void dog( double goat ) {
        System.out.print("Mammal "+goat);
    }
}
class Cat extends Mammal {
    public void dog( int cow ) {
        System.out.print("Cat "+cow);
    }
}
```

A. Mammal 5.0 Cat 3.0

B. Cat 3  Mammal 5.0

C. Cat 3  Cat 5.0

D. Mammal 3.0  Cat 5.0

# @override and @overload

- The @override and @overload statements are similar to comments
- They specify that the following method is overridden or overloaded
- If you put @override or @overload before a method that is not overridden or overloaded, you will get an error

# Polymorphism

- Subtype polymorphism allows a function to be written to take an object of a certain class `Parent`, but also work correctly if passed an object that belongs to a class `Child` that is a subclass of `Parent`
- Ad Hoc polymorphism allows functions to be overloaded so that the same function can take parameters of different types and perform differently

# Subtype Polymorphism Example

```
abstract class Animal {
  abstract String talk();
}
class Cat extends Animal {
  String talk() { return "Meow!"; }
 }
class Dog extends Animal {
  String talk() { return "Woof!"; }
 }
void letsHear(Animal goat) {
  System.out.println( goat.talk() );
}
public static void main( … ) {
  letsHear( new Cat() );
  letsHear( new Dog() );
}
```

# Three Pillars of OO Programming

- Inheritance
- Encapsulation
- Polymorphism