Inheritance

COMP360

“Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.”

Donald Knuth

Three Pillars of OO Programming

- Inheritance
- Encapsulation & Abstraction
- Polymorphism

Object Creation

- In Java, when you define an object of a class, it does not create the object until you use new
- When you use new, Java finds space on the heap for the object and copies the initial class image to the object
- Object creation starts with the class Object and continues down the inheritance hierarchy

```
Widget bird;          // no object created in Java
Widget bird;          // object created in C++
```

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)

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

```
public MyClass( ) {
    super();
}
```

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 is no longer available
- 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

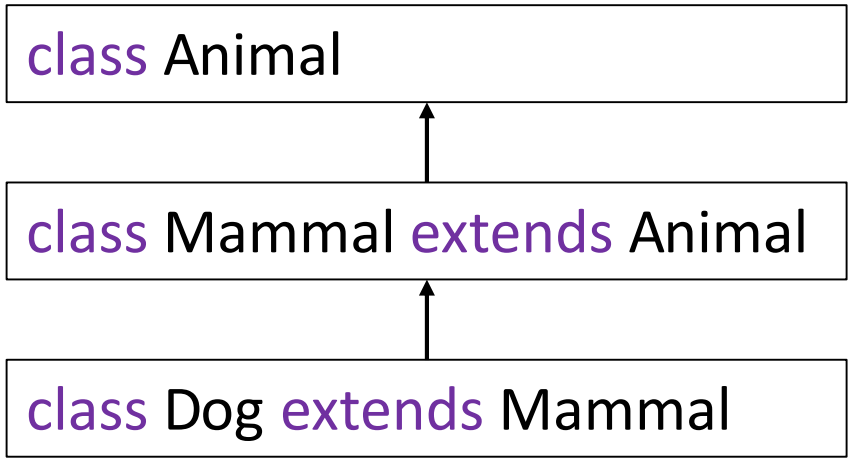
- One constructor can call another constructor of the same class using the method `this( … )`
- When used as a method name, `this` is the constructor for this class
- Call the this constructor is often used to provide default values when a parameter is not given

Constructors calling Constructors

```
public class Worm{
    public Worm() {
        this("default");
    }
    public Worm(String dirt) {
        System.out.print( dirt );
    }
}
Worm crawly = new Worm("fish");      // displays fish
Worm inch   = new Worm();            // displays default
```

Calling Constructor of all Ancestors

- Constructing an instance of a class invokes all the super classes' 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

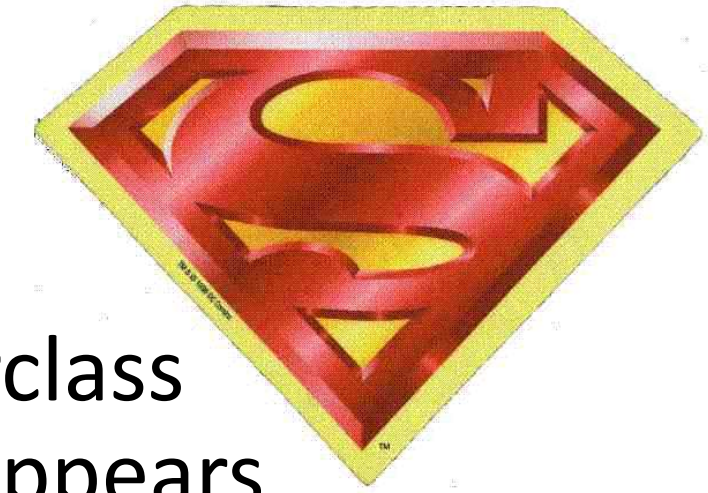
- Unlike all other methods, constructors are not inherited by subclasses
- If a class does not have a constructor, the constructor of the super classes will be called in order

What is displayed?

```
class Fish {
   public Fish() {
      this("Fish ");
      System.out.print("trout ");
   }
   public Fish(String minnow) {
      System.out.print("minnow");
      System.out.print(minnow);
   }
}
class Bass extends Fish {
   public Bass() {
      System.out.print("Bass ");
   }
}
public class ConOrder {
   public static void main(String[] x) {
      Bass swim = new Bass();
   }
}
```

A. Bass
B Fish trout Bass
C. trout Fish Bass
D. minnow Fish trout Bass
E. Fish trout minnow Bass

Using the Keyword `super`



- The keyword `super` refers to the superclass or parent 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 ) {
          super( cat ); // correct
          System.out.println("trout");
     }
}
```

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
     }
}
```

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
```

Avoiding Super

- The default super() constructor is implicitly called at the beginning of any constructor
- If you call super with a parameter, the default no parameter version will not be called
- If you call the this constructor, with or without parameters, the default super() will not be called
- The default super() might be called in the other constructor called by this(…)

Will this work?

```
public class Apple extends Fruit {
    int irrelevant = 0;
}
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

Default Constructor Needed

```
public class Apple extends Fruit {
    int irrelevant = 0;
}
class Fruit {
  public Fruit() {
    System.out.println("Default Fruit");
  }
  public Fruit(String name) {
    System.out.println("Fruit"+name);
  }
}
```

- Since Apple has no constructor, the default constructor calls a default constructor in the super class, Fruit

Default Constructor Needed

```
public class Apple extends Fruit {
   int irrelevant = 0;
   public Apple(String iName) {
      super(iName);
      System.out.println("Apple");
   }
}
class Fruit {
   public Fruit(String name) {
      System.out.println("Fruit"+name);
   }
}
```

- Apple calls the super constructor with a String parameter matching the Fruit constructor

Trace Execution

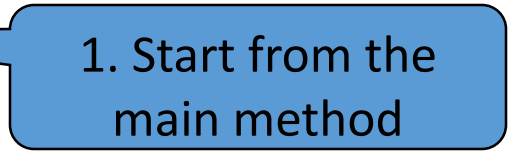
```
public class Trout extends Fish {
  public static void main(String[] args) {
    new Trout();
  }

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

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

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

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



Trace Execution

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

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

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

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

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

2. Invoke Trout constructor

Trace Execution

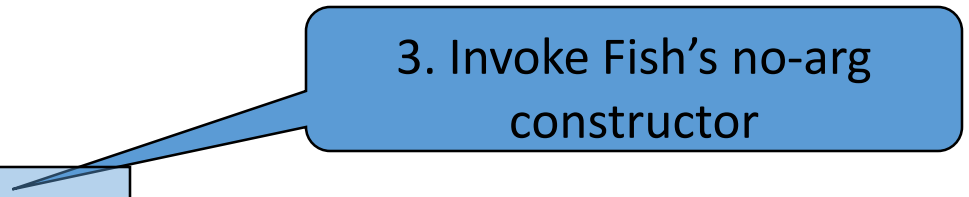
```
public class Trout extends Fish {
  public static void main(String[] args) {
    new Trout();
  }

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

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

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

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



Trace Execution

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

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

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

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

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

4. Invoke Fish(String) constructor

Trace Execution

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

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

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

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

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

5. Invoke Animal() constructor

Trace Execution

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

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

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

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

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

6. Execute println

Trace Execution

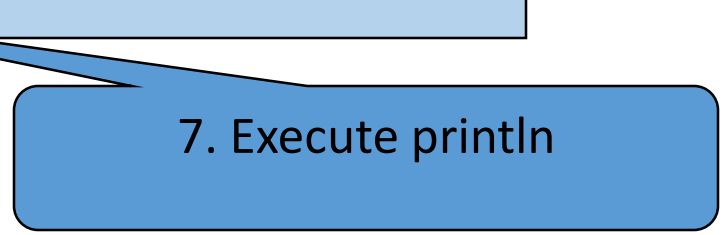
```
public class Trout extends Fish {
  public static void main(String[] args) {
    new Trout();
  }

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

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

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

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



Trace Execution

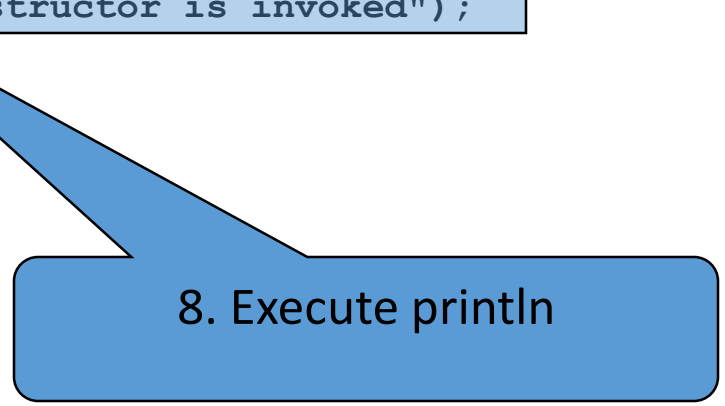
```
public class Trout extends Fish {
  public static void main(String[] args) {
    new Trout();
  }

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

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

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

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



Trace Execution

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

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

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

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

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

9. Execute println

Destructors

- Destructor functions are the inverse of constructor functions. They are called when an object is destroyed or deallocated
- In C++ a destructor method is always named the same as the class name preceded by a tilde, ~

```
MyClass :: ~MyClass() { … }
```

- They are useful is releasing resources that might not otherwise be released
- Java calls the destructor method `finalize`

```
public void finalize() { … }
```

Finalize Example

```
public class FinalDemo {
    private int myID;
    public static void main(String[] unused) {
        FinalDemo thing;
        for (int i = 0; i < 1000; i++) {
            thing = new FinalDemo(i);
        }
        System.out.println("Objects created");
        System.gc();     // force garbage collection
        System.out.println("All done");
    }
    public FinalDemo(int num) {
        myID = num;
    }
    public void finalize() {
        System.out.println("finalizing "+myID);
    }
}
```

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 & Abstraction
- Polymorphism

No class next week

Spring break is

March 5 to March 11, 2017