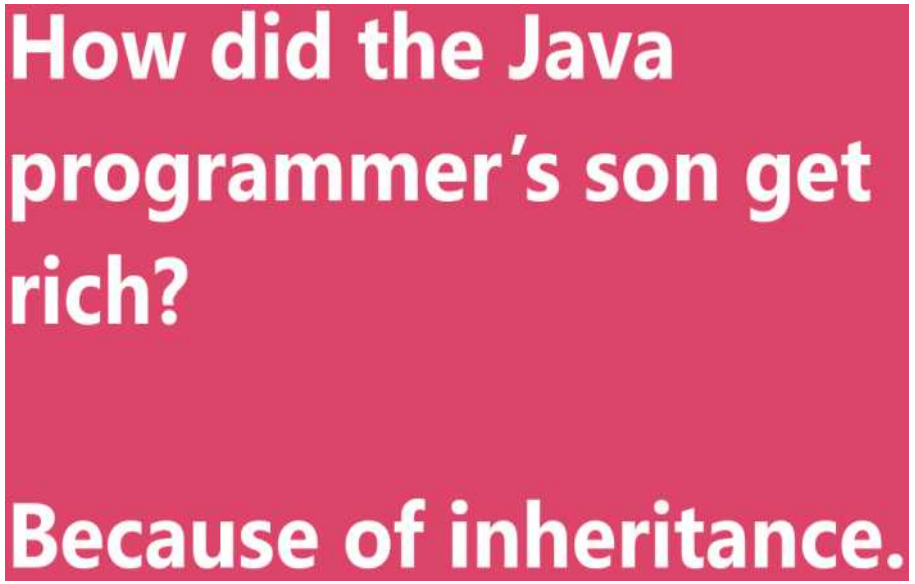
Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Inheritance



CS2012: Introduction to Programming II

Inheritance

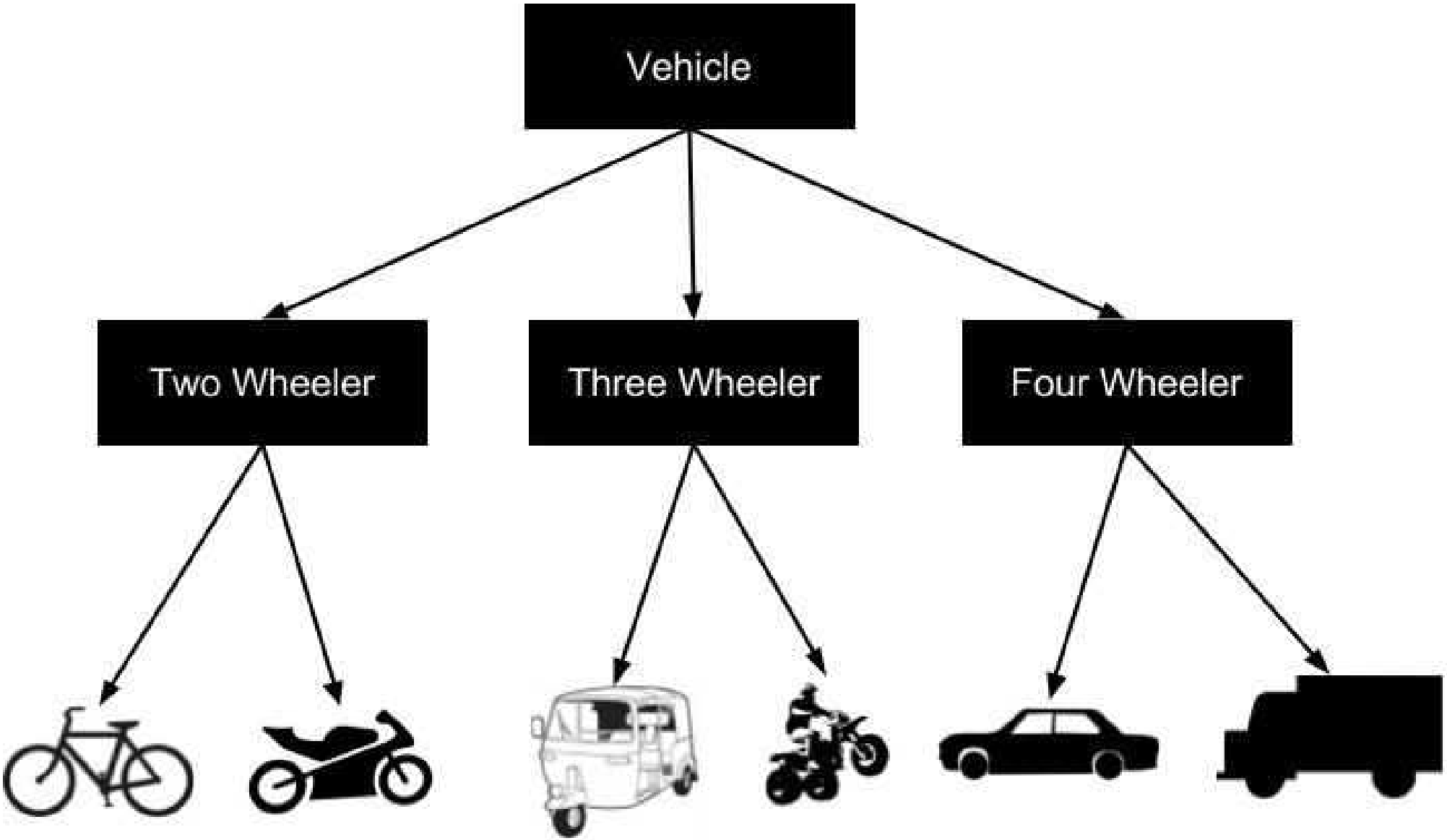
- ***inheritance*:**
  - one of the four pillars of OOP.
  - an important and powerful object oriented technique that allows one to define new classes from existing classes.
  - powerful feature for defining relationships between related classes and allowing a base class to reuse code written in the parent class.
  - defines an `is-a` relationship between two classes.

- Example:
  - Suppose you want to define a class to model various shapes (Circles, Rectangles, Triangles, etc.).
  - these classes could have many common features.
  - design the classes to avoid redundancy between them.

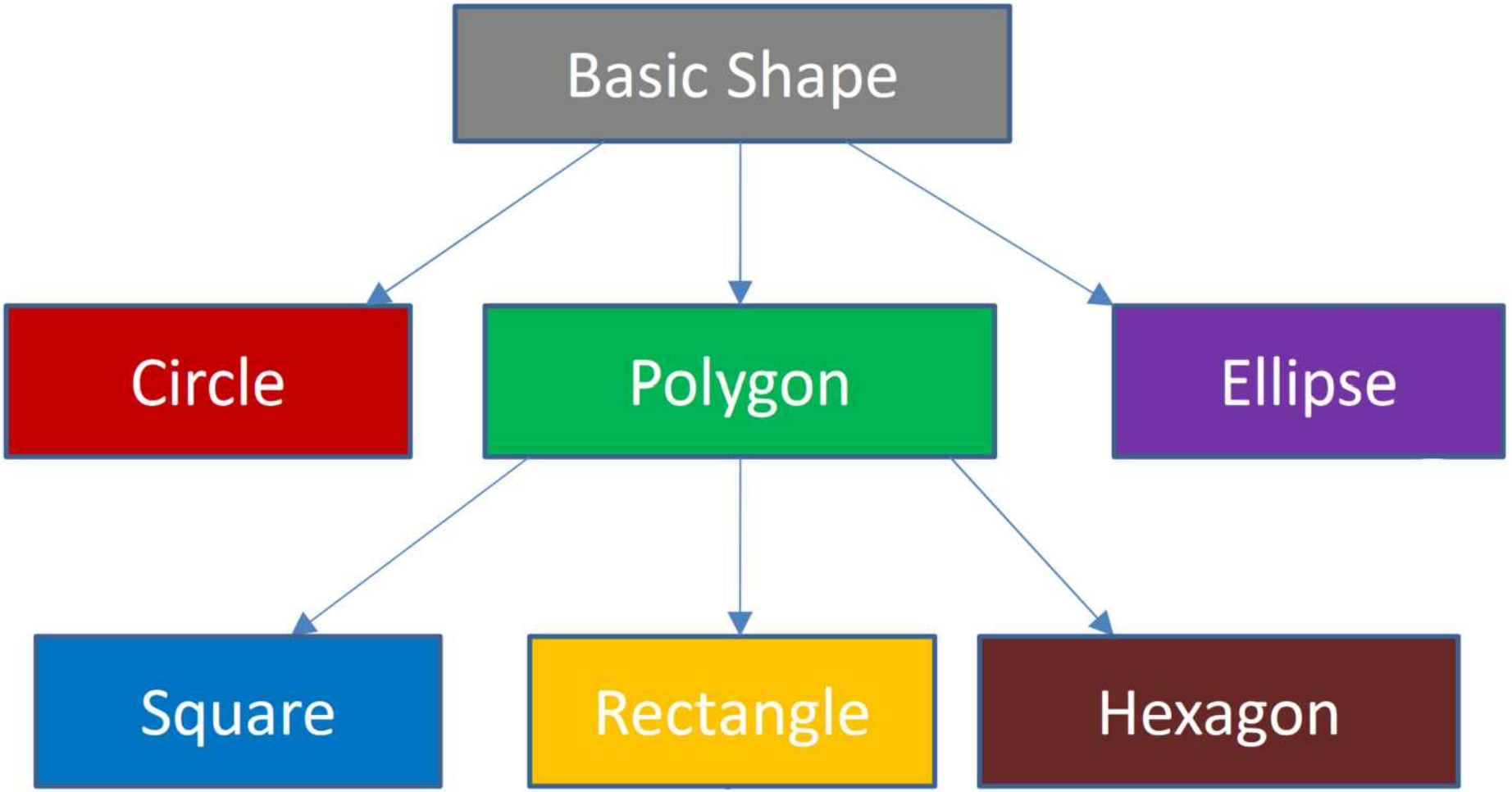
Inheritance

- Benefits of Inheritance:
  - Break software into more manageable pieces
  - Existing classes can be reused or reimplemented in new classes.
  - Different object types can be grouped together and actions can be performed on all of them at the same time.
  - Applications take less memory (less source code leads to less memory used once compiled.)
  - Redundancy (repetition) of the code is reduced or minimized so that we get more consistent results.

Inheritance Example



Inheritance Example



Superclasses and Subclasses

Superclasses and Subclasses

- You can use a class to model objects of the same type.

- Even if classes are different, they can share common properties and behaviors
  - these can be generalized in a class and then shared by other classes.

- Assume there are two classes C1 and C2 where C2 "extends" C1.
  - in this case C1 is the ***superclass***, ***parent class***, or ***base class***
  - and C2 is the ***subclass***, ***child class***, ***extended class***, or ***derived class***

- A subclass ***inherits*** all accessible data fields and methods from its superclass, and can even add new data fields and methods.

A First Example

- Let's design a class to model geometric objects like circles and triangles.

- Geometric objects have many common properties and behaviors.

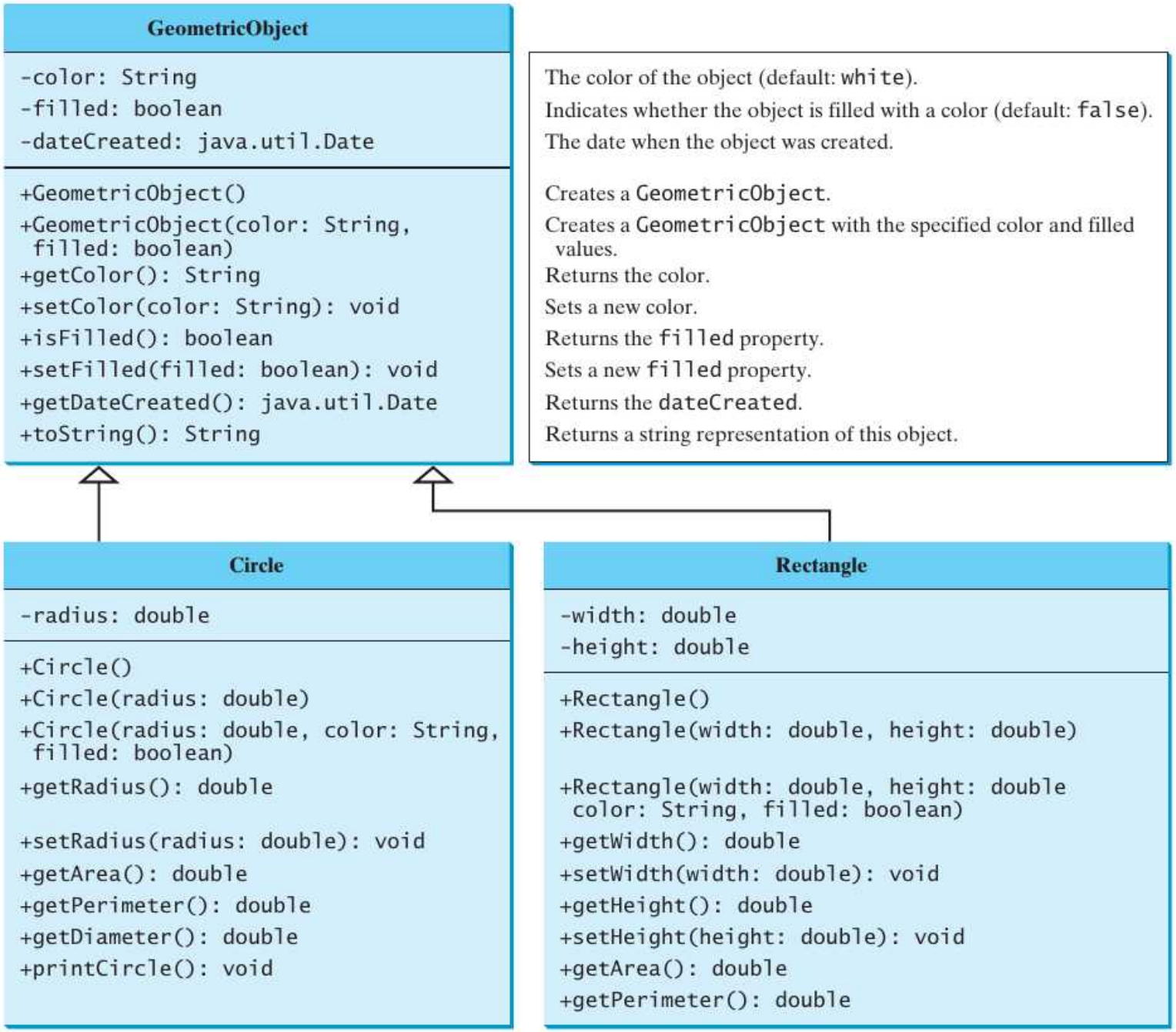
- `GeometricObject` Class:
  - a general class that can be used to model all geometric objects.
  - has properties `color` and `filled` and their getters / setters
  - also has `dateCreated` property and its getter
  - also has a `toString()` method to return a string representation of the object.

A First Example

- The `Circle` class:
  - a special type of geometric object
  - shares common properties and methods with other geometric objects
  - define `Circle` to "extend" the `GeometricObject` class
  - inherits everything from `GeometricObject`
  - adds some of its own methods.

- The `Rectangle` Class:
  - can also be defined as a subclass of `GeometricObject`
  - inherits everything from `GeometricObject`
  - adds some of its own methods.

A First Example



A First Example

- The `Circle` class uses the following syntax:

- extends is a keyword which tells the compiler that `Circle` is a subclass of `GeometricObject`, thus inheriting from the superclass.
- the constructor for `Circle` is implemented by invoking `setColor()`, and `setFilled()` which reside in the parent class inherited by Circle.
- You may be tempted to do something like this:

```
public CircleFromSimpleGeometricObject(
    double radius, String color, boolean filled) {
  this.radius = radius;
  this.color = color; // Illegal
  this.filled = filled; // Illegal
}
```

- However, `color` and filled are `private` in the `GeometricObject` class and are only accessible in that class (so not accessible to Circle)

Other Inheritance Facts

- Despite its naming, a subclass is not a subset of its superclass.
  - Usually a subclass contains more information and methods than its superclass.

- Private data fields in the superclass are never accessible outside of the class.
  - They cannot be used directly in a subclass.
  - They can be accessed / mutated through public getters / setters if these are define in the superclass.

- Not all `is-a` relationships should be modeled using inheritance
  - a square is a rectangle, but you should not extend `Square` from `Rectangle`
  - `width` and `height` properties are not appropriate for a square
  - better option is to define a Square class to extend GeomeotricObject and define a side property.

Other Inheritance Facts

- Inheritance models the is-a relationship
  - don't start extending classes all the time just to reuse methods.
  - it doesn't make sense for a `Tree` class to extend a Person class even if they share common properties like height and weight.
  - subclasses and superclasses should always have the is-a relationship.

- Two things to ask for whether or not to use inheritance?
  - do the two classes share common properties (data fields) and behaviors (methods)?
  - does the is-a relationship make sense in the real world?

Other Inheritance Facts

- Example 1: Tree extends Person
  - do the two classes share something in common
    - yes, they both have a height
  - does the is-a relationship make sense?
    - no, because a Tree is not a Person

- Example 2: Student extends Person
  - do the two classes share something in common
    - yes, they can have common properties
  - does the is-a relationship make sense?
    - yes, because a Student is-a Person

Other Inheritance Facts

- Unlike other languages, Java does not allow you to have ***multiple inheritance***
  - inheriting from more than one class
  - Java classes can only inherit directly from one superclass. (***single inheritance***)
  - (we will come back to multiple inheritance when we talk about interfaces.)

The super Keyword

The super Keyword

- `super` is a keyword which refers to the superclass of the class in which `super` is used

- `super` can be used in two ways:
  - call a superclass constructor.
  - call a superclass method.

Calling Superclass Constructors

- Constructors are not inherited by a subclass and can only be invoked using the `super` keyword.

- Examples:
  - `super();` //call the no-arg super constructor
  - `super(parameters);` //call the super constructor with the given parameters

- NOTE: the use of super to call the superclass constructor MUST be the first statement in the subclass constructor. invoking the superclass constructor with its class name in the subclass is a syntax error

Constructor Chaining

- A constructor can invoke an overloaded constructor or its superclass constructor. if neither is invoked explicitly, the compiler automatically puts `super()` as first statement in the constructor.

```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(double d) {
  // some statements
}
```

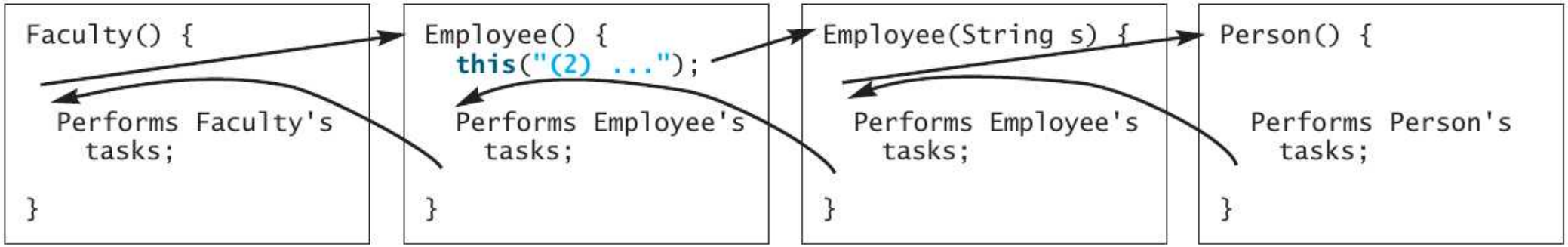
Equivalent

```
public ClassName(double d) {
  super();
  // some statements
}
```

- Constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain.
- subclass constructors always invoke the superclass constructors before performing their own tasks even if there are multiple levels of inheritance

Constructor Chaining Example

- See Code: Faculty.java, Employee.java, and Person.java



Constructor Chaining Caution

- Classes should always define a no-arg constructor to avoid programming errors. Consider:

```
1  public class Apple extends Fruit  {
2  }
3
4  class Fruit  {
5    public Fruit(String name) {
6      System.out.println("Fruit's constructor is invoked");
7    }
8  }
```

- Why does this cause compile errors?

Calling Superclass Methods

- `super` can be used to call other methods from the superclass.

- Syntax:
  - `super.method(parameters);`

- `printCircle()` from the `Circle` class could be rewritten as follows:

```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

Caution: No ~~super.super.methodName()~~

- Assume there are three classes A, B, and C.
- Assume C inherits from B and B inherits from A
  - A ► B ► C
- Assume A has a method `myMethod()` which is implemented in B and C
- Question: if you can use `super.myMethod()` in B to call A's version of `myMethod()`, can you also use `super.super.myMethod()` to call A's version of `myMethod()` in C?
  - No, this is a syntax error.
  - Only allowed one level of method chaining with super.

Overriding Methods

Overriding Methods

- ***override a method***: when a subclass modifies the implementation of a method defined in the super class
  - you have already seen this when you "override" the `toString()` method.
  - `toString()` is defined in the `Object` class, and can be given a more specific implementation in any class in Java (since all classes are subclasses of `Object`).

Overridding Methods

- Rules for Overridding:
  - instance methods can only be overridden if it is accessible (public). if a method defined in the subclass is private in the superclass, the two methods are unrelated.
  - static methods can be inherited, but a static method cannot be overridden
    - if a static method defined in the superclass is redefined in a subclass, the superclass version is "hidden". to access it you need to use `SuperClassName.staticMethodName`

Overriding vs Overloading

Overriding vs. Overloading

- ***overloading***: define multiple methods with the same name, but different method signatures.
  - overloaded methods can be either in the same class or different classes related by inheritance
  - overloaded methods have the same name but different parameter lists

- ***overriding***: providing a completely new implementation for a method in a subclass.
  - overridden methods are in different classes related by inheritance
  - overridden methods have the exact same signature and return type.

Overriding vs. Overloading

```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

(a)

```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

(b)

Override Annotation

- To avoid any confusion / mistakes, you can use a special Java syntax called the override annotation to indicate that a method is an override of another method.
  - annotation is `@Override` and is placed before the method in the subclass.
  - the annotation denotes that the annotated method is required to override a method in the superclass
  - if the method with this annotation does not override it's superclass method, the compiler will report an error.

- Example: if `toString()` is mistyped `tostring()` a compile error is reported if the annotation is present, no compile error reported if the annotation is omitted.

Override Annotation Example

```
1  public class CircleFromSimpleGeometricObject
2      extends SimpleGeometricObject {
3    // Other methods are omitted
4
5    @Override
6    public String toString() {
7      return super.toString() + "\nradius is " + radius;
8    }
9  }
```

Override Annotation Example

```
1  public class CircleFromSimpleGeometricObject
2      extends SimpleGeometricObject {
3    // Other methods are omitted
4
5    @Override
6    public String toString() {
7      return super.toString() + "\nradius is " + radius;
8    }
9  }
```

References

- Liang, Chapter 11