Inheritance and Composition

Dr. Youna Jung

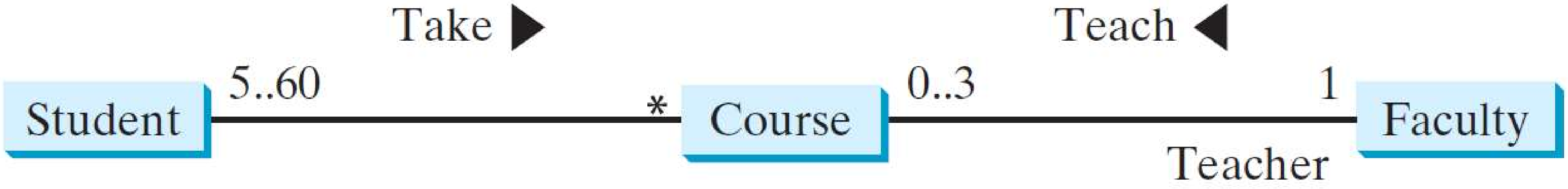
Northeastern University

yo.jung@northeastern.edu

LVX
VERITAS
VIRTVS

Class **Relationships**

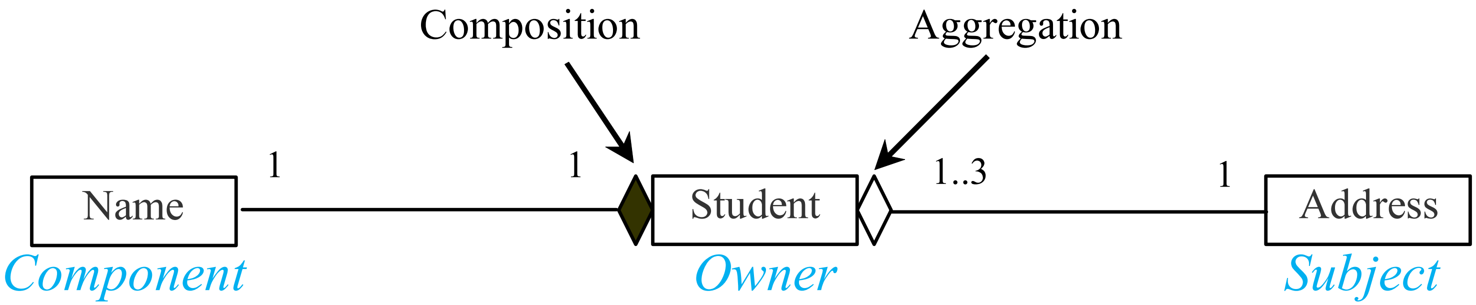
- □ **Association**
  - ✓ A **general binary relationship** that describes an activity between two classes.
- □ **Composition**
- □ **Aggregation**
- □ **Inheritance**



COMPOSITION

Aggregation VS Composition

- **Aggregation** (*has-a* relationships)
  - ✓ represents an **ownership** relationship between two objects
    - The **owner** class/object
      - ➢ ***Aggregating*** *object* and ***Aggregating*** *class*
    - The **subject** class/object
      - ➢ ***Aggregated*** *object* and its class an ***Aggregated*** *class*.
- **Composition**
  - ✓ A **special case** of the **aggregation** relationship
    - If the **owner cannot exist without subject**



Aggregation

- An aggregation relationship is usually represented as **a data field** in the **owner class**

```
public class Name {
  ...
}
```

Aggregated class

Subject

```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

Aggregating class

Owner

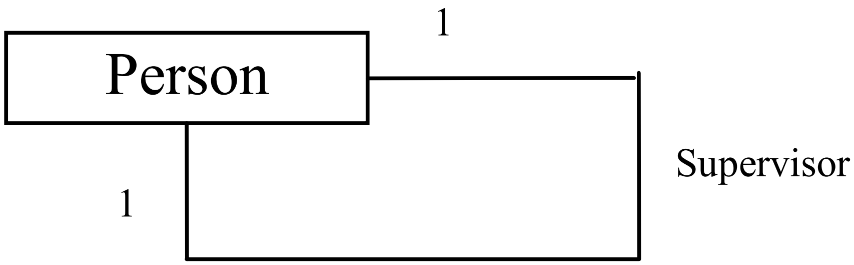
```
public class Address {
  ...
}
```

Aggregated class

Subject

Aggregation Between Same Class

- Aggregation may exist between objects of the same class
  - ✓ E.g.) A person may have a supervisor.

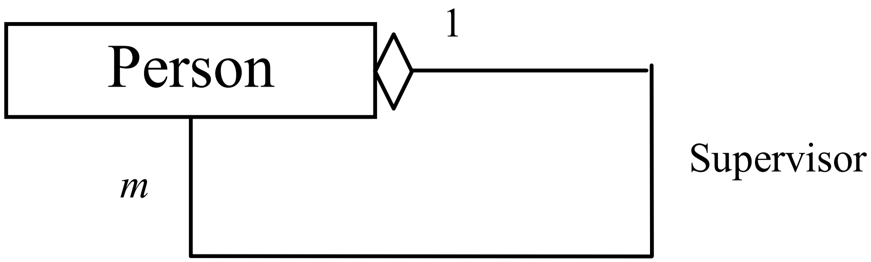


```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;
  …
}
```

INHERITANCE

Aggregation Between Same Class

What happens if a person has several supervisors?



```
public class Person {
  ...
  private Person[] supervisors;
}
```

Inheritance

- ❑ Suppose you will define **classes** to model **circles**, **rectangles**, and **triangles**.
  - ✓ These classes have many **common features** (*e.g. they can be drawn in a certain color and be filled or unfilled*). What is the best way to design these classes so to **avoid redundancy**?

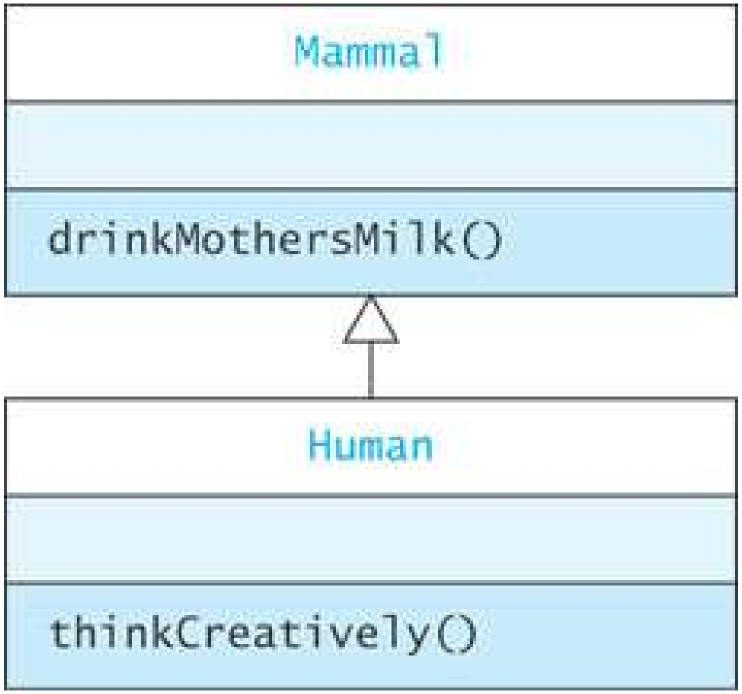
↓

use **Inheritance**!

- enables you to define a **general** class (***superclass***) and later **extend** it to more **specialized** classes (***subclasses***)

Inheritance

- A **Human** *is a* **Mammal**
  - Mammal is the **superclass** of Human
    - Mammal has only method **`drinkMothersMilk()`**
    - Human has all the data fields and methods defined by Mammal
  - Human is a **subclass** of Mammal
    - Human may define other variables and methods that are not contained in Mammal
    - Human has method **`drinkMothersMilk()`** and **`thinkCreatively()`**



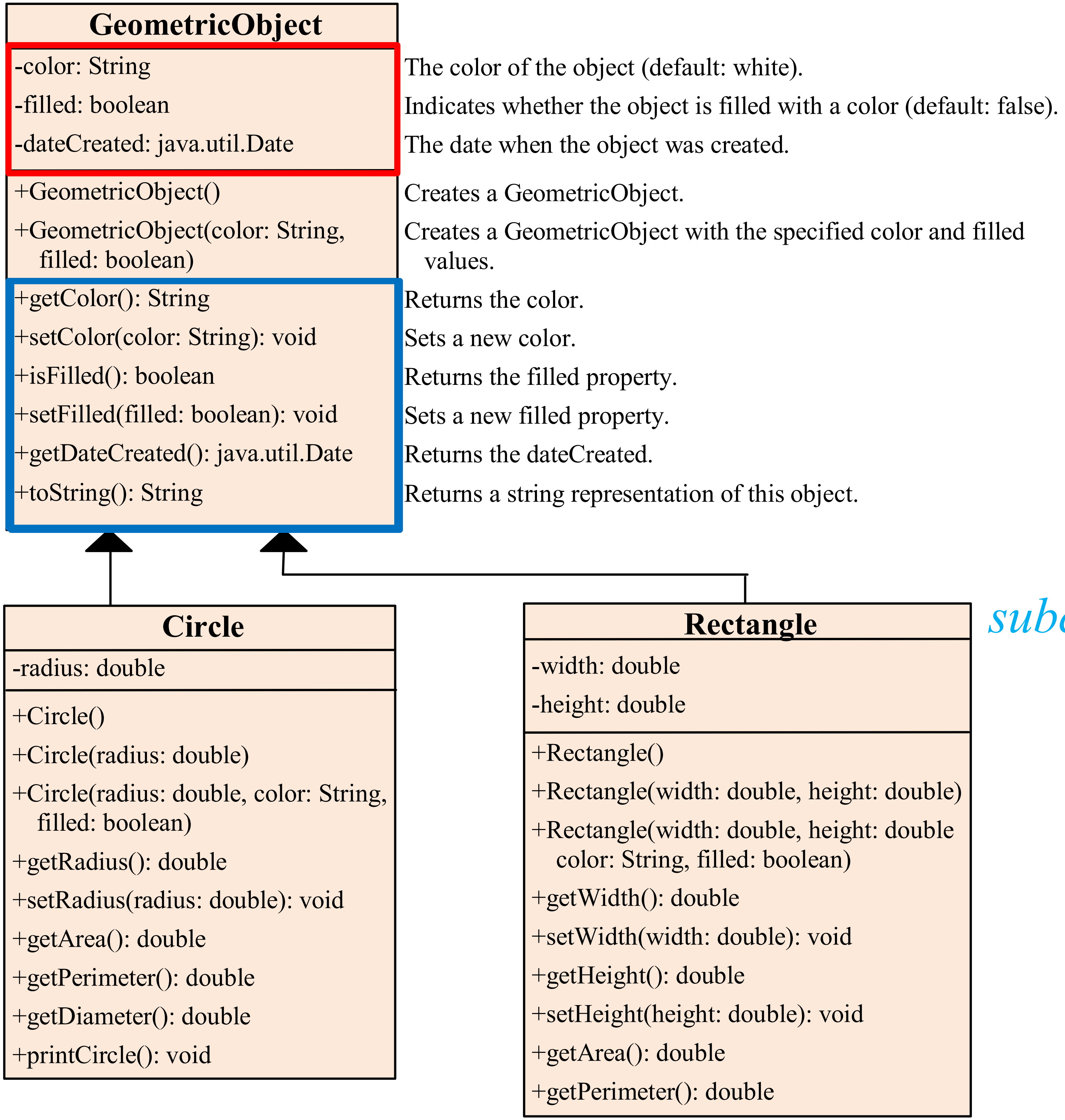
Inheritance

- ❑ The **way** to **define new classes** from **existing** classes (reusing software)
  - ✓ used to model the ***is-a*** relationship
  - ✓ Java does not allow ~~*multiple inheritance*~~
- ❑ A class `C1` extended from another class `C2`.
  - ✓ `C2` is called a ***superclass*** (*parent* or *base* class)
  - ✓ `C1` is called a ***subclass*** (*child/extended/derived* class)
    - – **inherits** accessible **data fields** and **methods from** its superclass (inheritance)
      - ➢Only accessible members
        - → `private` members cannot be inherited!
        - → can be accessed through `public` **accessor** or **mutator**
    - – AND may also **add** new data fields and methods (**extension/specialization**)

superclass

Private member →
Cannot be inherited

Public member →
Inherited to subclass



GeometricObject

-color: String — The color of the object (default: white).

-filled: boolean — Indicates whether the object is filled with a color (default: false).

-dateCreated: java.util.Date — The date when the object was created.

+GeometricObject() — Creates a GeometricObject.

+GeometricObject(color: String, filled: boolean) — Creates a GeometricObject with the specified color and filled values.

+getColor(): String — Returns the color.

+setColor(color: String): void — Sets a new color.

+isFilled(): boolean — Returns the filled property.

+setFilled(filled: boolean): void — Sets a new filled property.

+getDateCreated(): java.util.Date — Returns the dateCreated.

+toString(): String — Returns a string representation of this object.

subclass

Circle

-radius: double

+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

subclass

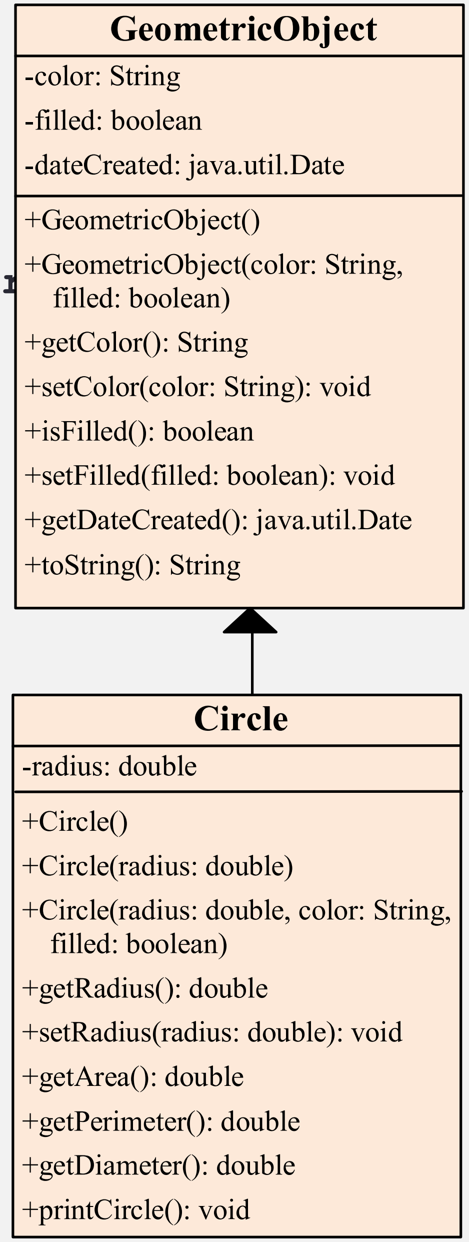
```
public class SimpleGeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

public SimpleGeometricObject() {
    dateCreated = new java.util.Date();    }
public SimpleGeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;   }

  public String getColor() {
    return color;   }

  public void setColor(String color) {
    this.color = color;   }

  public boolean isFilled() {
    return filled;   }

  public void setFilled(boolean filled) {
    this.filled = filled;   }

  public java.util.Date getDateCreated() {
    return dateCreated;   }

  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;   }    }
```

```
public class CircleFromSimpleGeometricObject extends SimpleGeometricObject {
  private double radius;

public CircleFromSimpleGeometricObject() {
}
public CircleFromSimpleGeometricObject(double radius) {
    this.radius = radius;   }
public CircleFromSimpleGeometricObject(double radius, String colo
```

```
  super(color,filled);
  this.radius = radius;
```

```
s.color = color;     // Error!
s.filled = filled;   // Error!
```

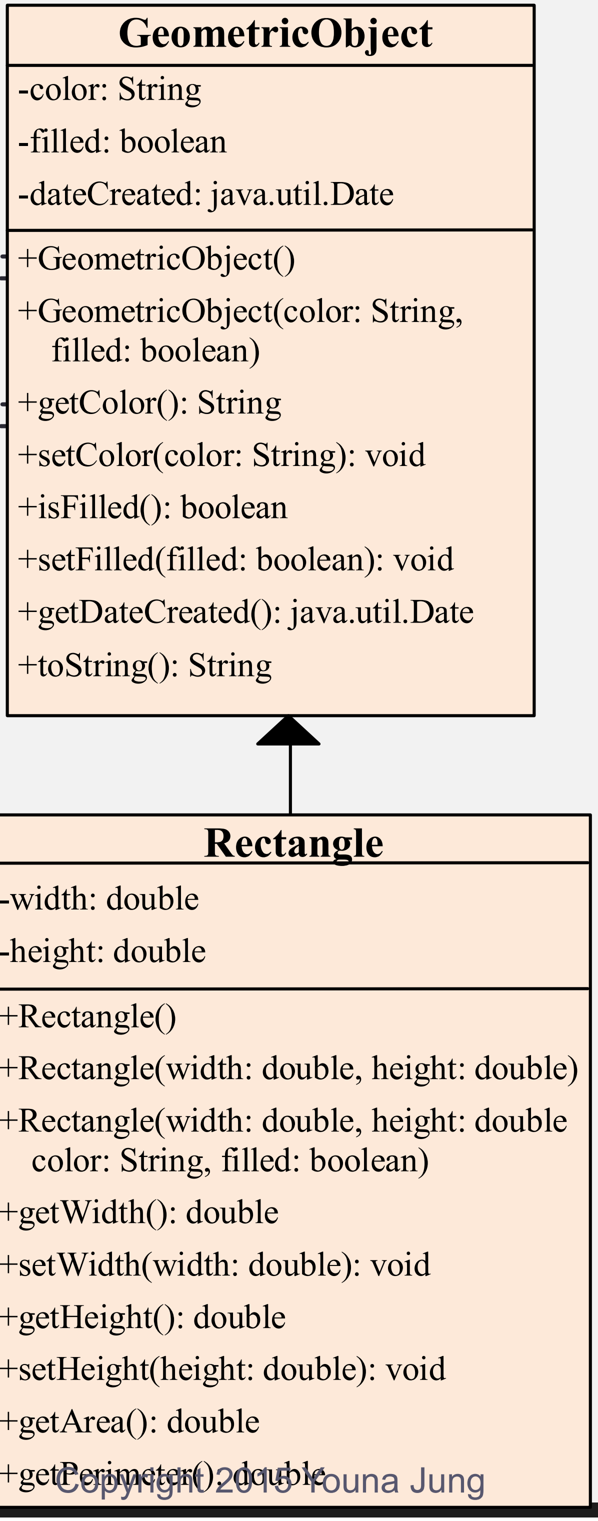
```
  public double getRadius() {
    return radius;  }
  public void setRadius(double radius) {
    this.radius = radius;   }

  public double getArea() {
    return radius * radius * Math.PI;   }

  public double getDiameter() {
    return 2 * radius;   }

  public double getPerimeter() {
    return 2 * radius * Math.PI;   }

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



GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String

Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void

```
public class RectangleFromSimpleGeometricObject extends SimpleGeometricObject {
  private double width;
  private double height;

public RectangleFromSimpleGeometricObject() {
}
public RectangleFromSimpleGeometricObject(double width, doubl
    this.width = width;
    this.height = height; }
public RectangleFromSimpleGeometricObject(double width, doubl
        String color, boolean filled) {
    this.width = width;
    this.height = height;
    setColor(color);
    setFilled(filled);  }

  public double getWidth() {
    return width;  }
  public void setWidth(double width) {
    this.width = width; }

  public double getHeight() {
    return height;  }
  public void setHeight(double height) {
    this.height = height;  }

  public double getArea() {
    return width * height;  }

  public double getPerimeter() {
    return 2 * (width + height);  }}
```



GeometricObject

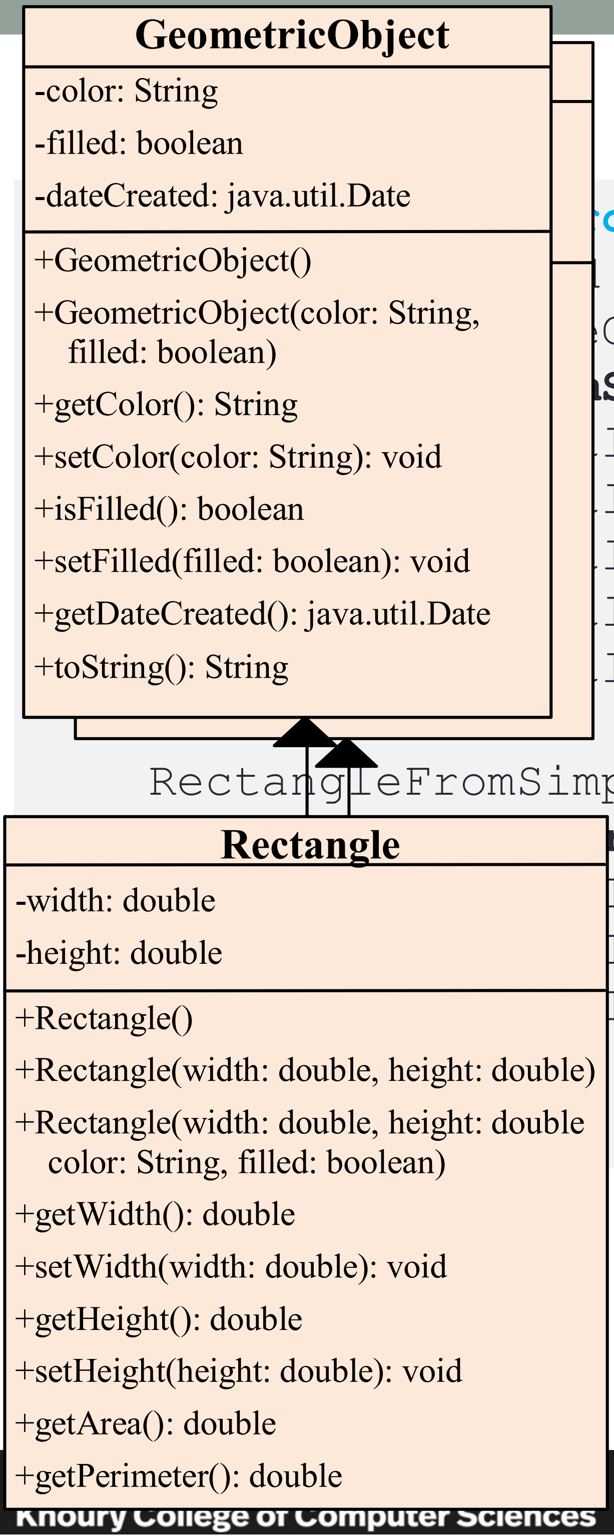
-color: String
-filled: boolean
-dateCreated: java.util.Date

+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double



GeometricObject

-color: String
-filled: boolean
-dateCreated: java.util.Date

+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

```
cleRectangle {
l main(String[] args) {
GeometricObject circle =
SimpleGeometricObject(1);
ln("A circle " + circle.toString());
ln("The color is " + circle.getColor());
ln("The radius is " + circle.getRadius());
ln("The area is " + circle.getArea());
ln("The diameter is " + circle.getDiameter());

    RectangleFromSimpleGeometricObject rectangle =
romSimpleGeometricObject(2, 4);
ln("\nA rectangle " + rectangle.toString());
ln("The area is " + rectangle.getArea());
ln("The perimeter is " + rectangle.getPerimeter());
```

Keyword `super`

- refers to the **superclass** of the class in which `super` appears. This keyword can be used in two ways:

  1) To call a **superclass constructor**
  2) To call a **superclass method**

Are Superclass's Constructor Inherited?

- ❑ **No**! A **superclass's constructors** are **not inherited** in the subclass.
  - ✓ BUT they can be **invoked** explicitly or implicitly.
    - – Explicitly using the **`super`** keyword.

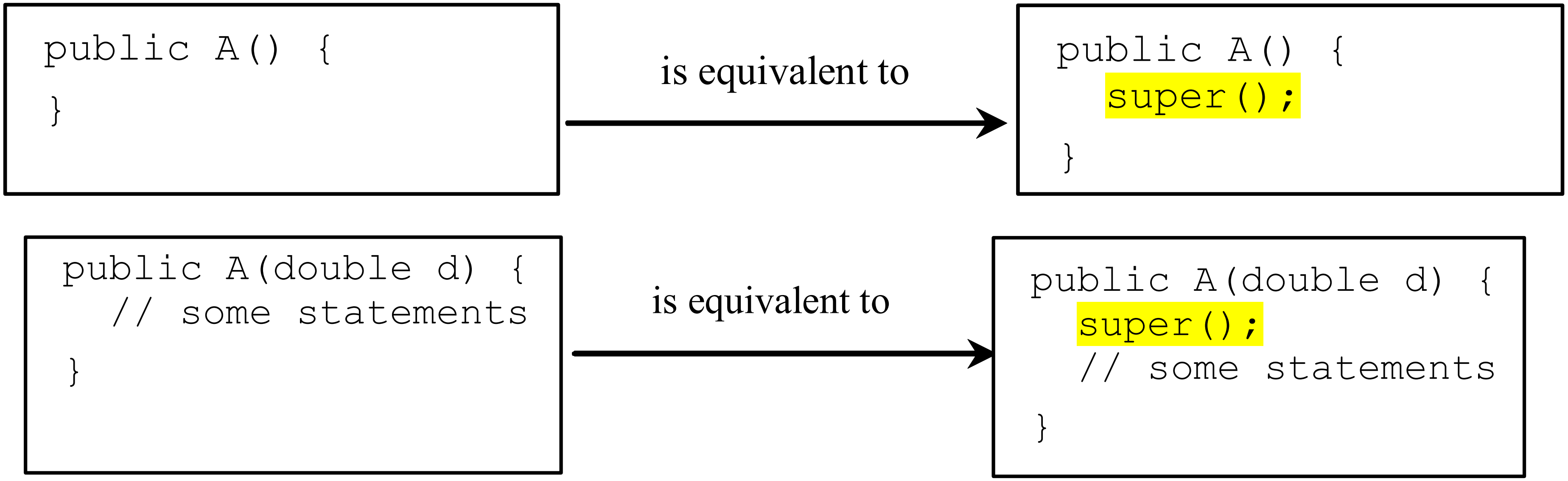
```
super()            // invokes the no-arg constructor of its superclass
super(parameters)  // invokes the superclass constructor matched
```

- ➢Caution: `super()` or `super(para)` **must** be the **first** statement of the **subclass's constructor** !!
- ➢Caution: ~~**Invoking a superclass constructor by method name** in~~ a subclass causes a **syntax error** !!
- ➢Caution: can invoke `super()` or `super(para)` **just one time**!

- – If the keyword **`super`** is not explicitly used → the **superclass's *no-arg*** *constructor* is **automatically invoked**.

Constructor Chaining

- When constructing an **object** of a **subclass**, the subclass constructor **first** invokes its **superclass constructor before** performing **its own** tasks
  - ✓ ➔ In any case, a constructor invokes the constructors of all the superclasses along the inheritance chain (***constructor chaining***)



```
public A() {
}
```

is equivalent to

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

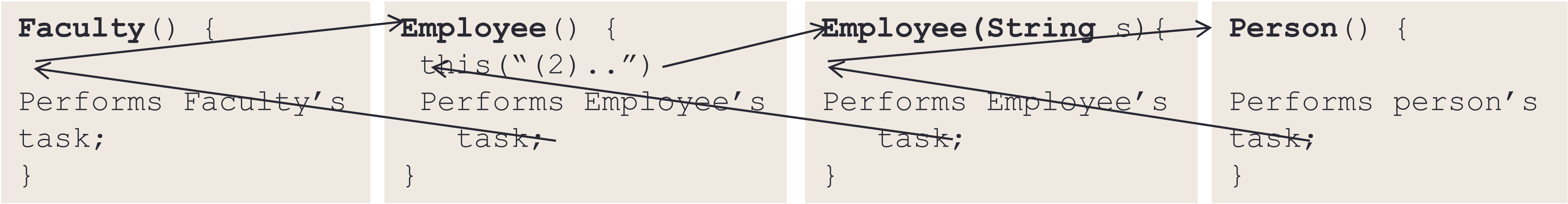
```
public A(double d) {
  // some statements
}
```

is equivalent to

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

```
public class Faculty extends E
  public static void main(Stri
    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) Person's no-arg constructor is invoked
(2) Invoke Employee’s overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty's no-arg constructor is invoked
```



```
Faculty() {

Performs Faculty’s
task;
}
```

```
Employee() {
  this(“(2)..”)
  Performs Employee’s
    task;
}
```

```
Employee(String s){

Performs Employee’s
    task;
}
```

```
Person() {

Performs person’s
task;
}
```

Superclass without no-arg Constructor

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

- No constructor is explicitly defined in **`Apple`** → **`Apple`**’s default no-arg constructor is defined implicitly
  - Since `Apple` is a subclass of **`Fruit`**, `Apple`’s default constructor automatically invokes **`Fruit`**’s **no-arg constructor**
- But **`Fruit`** does not have a no-arg constructor

→ **Compile Error!!**

Defining a Subclass

- A subclass **inherits accessible data fields** and **methods** from a **superclass**. In addition, you can also
  - ✓ Add **new data** fields
  - ✓ Add **new methods**
  - ✓ **Override** the methods of the superclass

Calling **Superclass Methods**

- ❑ You could write the **`printCircle`**`()` method in the **`Circle`** class using the **method** of **its super class** as follows:

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

Method Overriding

- ❑ Sometimes it is necessary for the **subclass** to **modify** the implementation of a **method** defined in the **superclass**.
  - ✓ To override a method, the **method** must be defined in the **subclass using <u>the same signature</u> and <u>the same return type</u>** as in its **superclass**.

```
public class Circle extends GeometricObject {

  // Other methods are omitted

  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;  }

}
```

NOTE

- ❑ An **instance** 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.

NOTE

- Like an instance method, a `static` method can be **inherited**.

- However, a `static` method **cannot be overridden**.
  - ✓ If a `static` method defined in the superclass is **redefined** in a **subclass**, the method defined in the **superclass** is **hidden**.

Overriding vs. Overloading

❑ Overloading

- ✓ means to define **multiple methods** with the ***same name*** but ***different signature***.
- ✓ Overloaded methods can be **either in** the ***same class*** or ***different classes*** related by inheritance

❑ Overriding

- ✓ have the ***same signature*** *and* the ***same return type***
- ✓ means to provide a **new implementation** for a method in the **subclass**.
- ✓ Overridden methods **must** be **in *different classes*** related by **inheritance**.
  - – Overridden method in a **superclass**
  - – Overriding method in a **subclass**

Overriding vs. Overloading

```
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);
  }
}
```

```
10.0
10.0
```

```
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);
  }
}
```

```
10
20.0
```

`@Override`

- ❑ Override annotation (**`@Override`**)
  - ✓ a special annotation, denotes that the annotated method is required to **override** a method **in the superclass**
    - If a method with **`@Override`** does not override its superclass's method → **Compile Error**!
    - Without `@Override`, cannot catch a mistake.

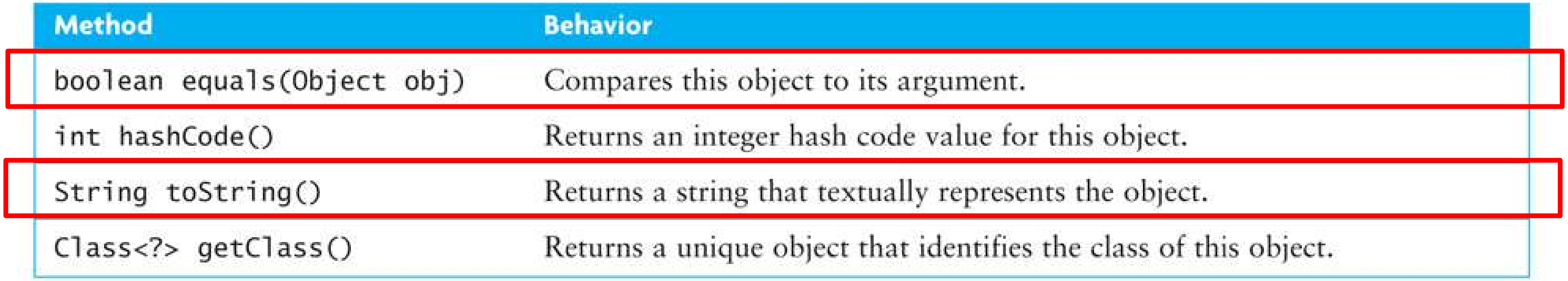
Object Class

- Every class in Java is descended from the `java.lang.Object` class.
  - ✓ If no inheritance is specified when a class is defined, the superclass of the class is `Object`.

```
public class Circle {
   ...
}
```

Equivalent

```
public class Circle extends Object {
   ...
}
```



Method	Behavior
`boolean equals(Object obj)`	Compares this object to its argument.
`int hashCode()`	Returns an integer hash code value for this object.
`String toString()`	Returns a string that textually represents the the object.
`Class<?> getClass()`	Returns a unique object that identifies the class of this object.

`toString()` in `Object` class

- ❑ returns a **string representation** of the **object**.
  - ✓ The default implementation returns a string consisting of 1) a **class name** of which the object is an instance, 2) the at sign (**@**), and 3) the **object's memory address** in hexadecimal.

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

```
Loan@15037e5
```

– Usually you should override the `toString()` method so that it returns a digestible string representation of the object.

The equals() Method

- The equals() method compares the **contents** of two **objects**.
  - ✓ The default implementation of the equals() method in the Object class is as follows:

```
public boolean equals(Object obj) {
     return this == obj;
}
```

  - ✓ The equals() method is overridden in the Circle class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;}
  else
    return false;}
```

`==` vs `equals()`

- ❑ The `==` comparison operator
  - ✓ for comparing two **primitive data type values** OR
  - ✓ for determining whether two objects have the **same references**.

- ❑ `equals()`
  - ✓ test whether two objects have the same **contents**, provided that the method is modified in the defining class of the objects.

Comparable Interface

- Classes that implement the `Comparable` interface must define a `compareTo()` method
  - Implementing the `Comparable` interface is an efficient way to compare objects during a search
  - Method call `obj1.compareTo(obj2)` returns an integer with the following values
    - **negative** if `obj1 < obj2`
    - 0 if `obj1 == obj2`
    - positive if `obj1 > obj2`

ENUMERATED TYPE

Enumerated Type

- A special class
  - An enumerated type variable is a reference variable.
- Defines a list of enumerated values
  - Each value is an identifier

```
enum MyFavoriteColor {RED, BLUE, GREEN, YELLOW};
```

    - Declared type = MyFavoriteColor, Values = RED, BLUE, GREEN, YELLOW
- An enumerated type is named like a class
  - with first letter of each word capitalized.
- A value of an enumerated type is like a constant
  - By convention, is spelled with all uppercase letters

Enumerated Type

- Once a type is defined, you can declare a variable of that type

```
MyFavoriteColor color;
```

  - ✓ **`color`** can hold one of the values defined in **`MyFavoriteColor`**

- The enumerated values can be accessed using the syntax

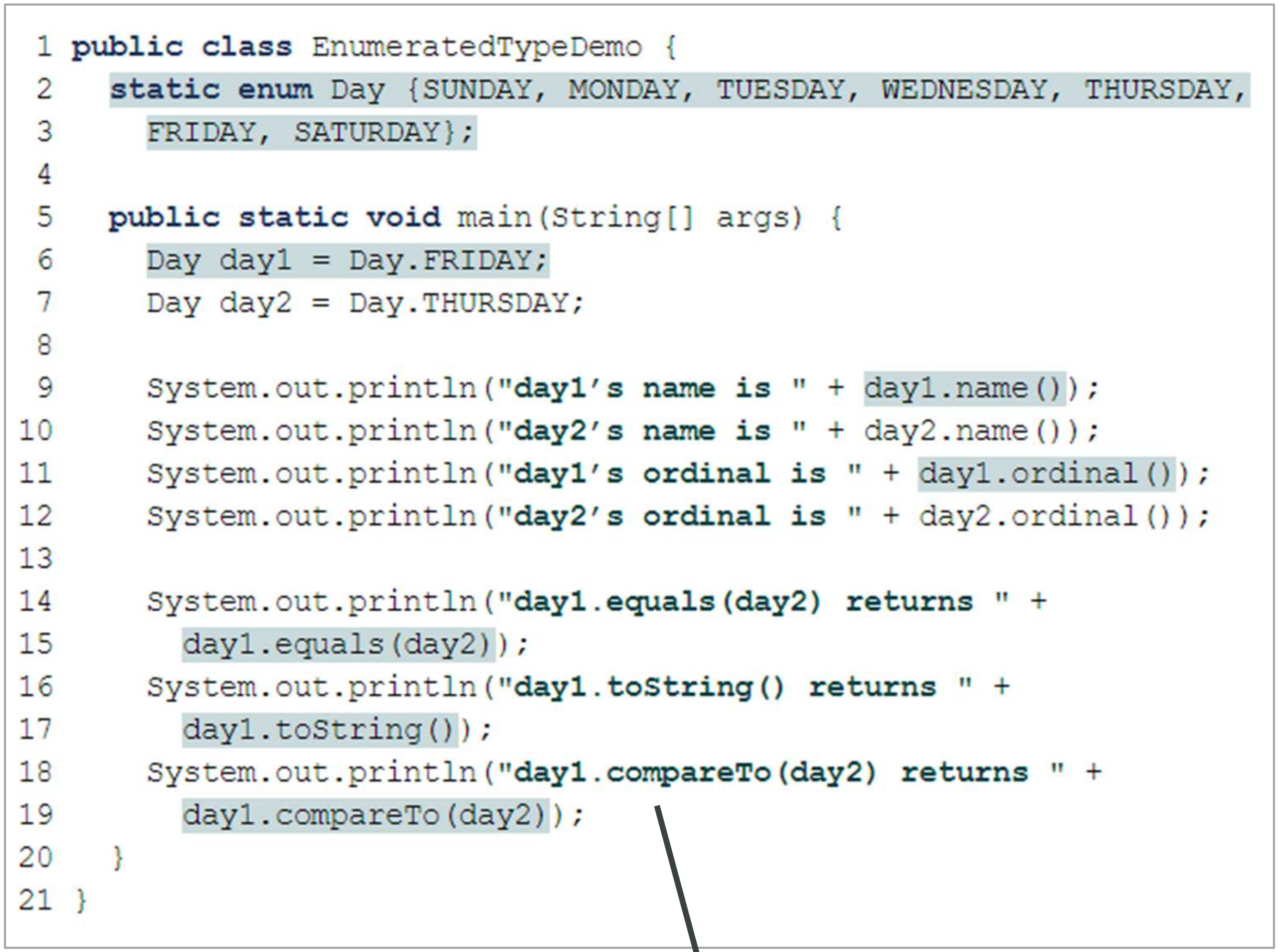
```
EnumeratedTypeName.valueName;
```

```
color = MyFavoriteColor.BLUE;
```

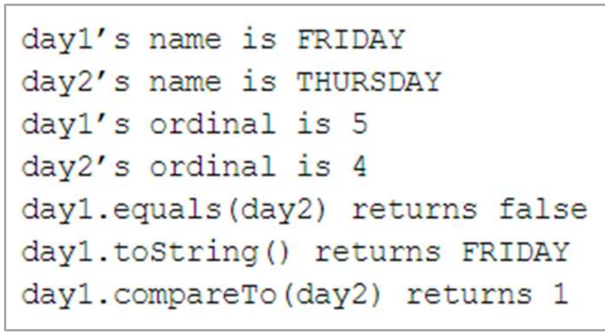
  - ✓ assigns enumerated value **BLUE** to variable `color`:

Enumerated Type

- ❑ An enumerated type is a subtype of the `Object` class and the `Comparable` interface
  - ✓ inherits all the methods in the Object class and `compareTo()` method in the Comparable interface
  - ✓ Additionally, you can use the following methods on an enumerated object
    - `public String name();`
      - ➢ Returns a name of the value for the object.
    - `public int ordinal();`
      - ➢ Returns the ordinal value associated with the enumerated value
        - ✓ The first value in an enumerated type has an ordinal value of 0
        - ✓ The second has an ordinal value of 1
        - ✓ The third one 3, and so on.



```java
public class EnumeratedTypeDemo {
  static enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY};

  public static void main(String[] args) {
    Day day1 = Day.FRIDAY;
    Day day2 = Day.THURSDAY;

    System.out.println("day1's name is " + day1.name());
    System.out.println("day2's name is " + day2.name());
    System.out.println("day1's ordinal is " + day1.ordinal());
    System.out.println("day2's ordinal is " + day2.ordinal());

    System.out.println("day1.equals(day2) returns " +
      day1.equals(day2));
    System.out.println("day1.toString() returns " +
      day1.toString());
    System.out.println("day1.compareTo(day2) returns " +
      day1.compareTo(day2));
  }
}
```

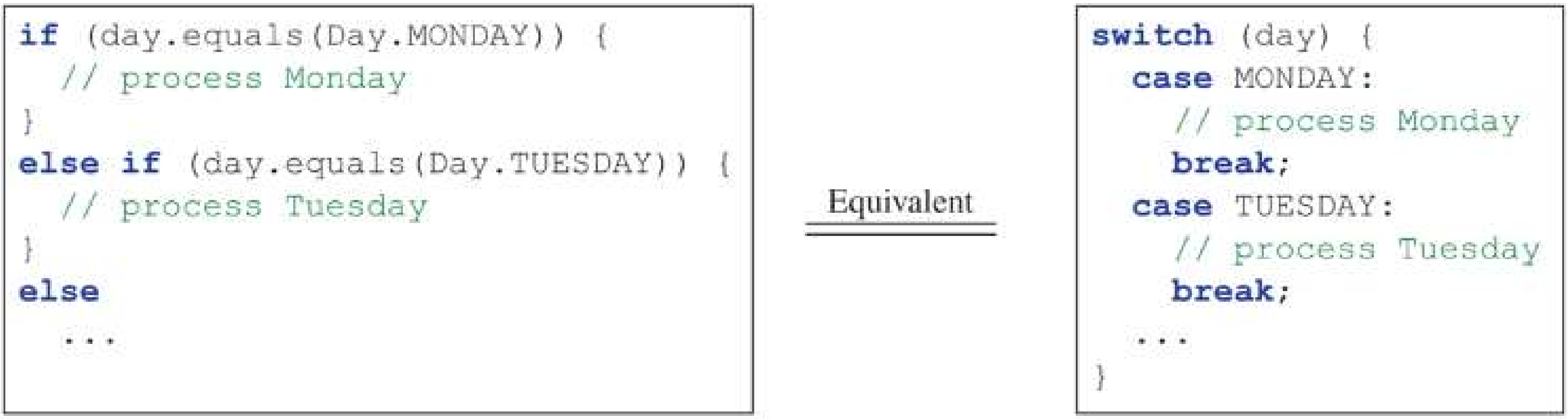


```
day1's name is FRIDAY
day2's name is THURSDAY
day1's ordinal is 5
day2's ordinal is 4
day1.equals(day2) returns false
day1.toString() returns FRIDAY
day1.compareTo(day2) returns 1
```

`day1.compareTo(day2)` returns the difference between day1's ordinal value and day2's.

`If` or `switch` with `Enum` Variables

- You can use an `if` statement or a `switch` statement to test the value in the variable



```
if (day.equals(Day.MONDAY)) {
  // process Monday
}
else if (day.equals(Day.TUESDAY)) {
  // process Tuesday
}
else
  ...
```

Equivalent

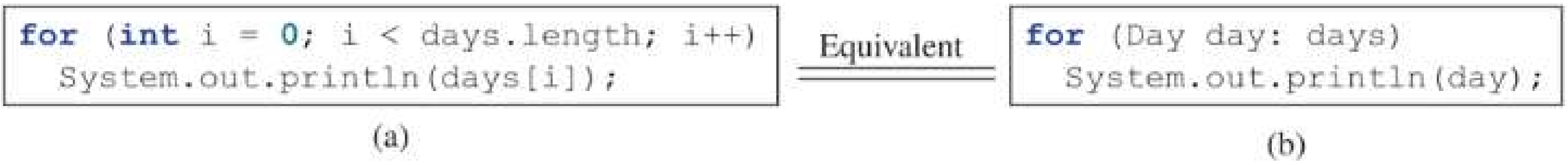
```
switch (day) {
  case MONDAY:
    // process Monday
    break;
  case TUESDAY:
    // process Tuesday
    break;
  ...
}
```

Loop with **Enum** Variables

- ❑ Each enumerated type has a static method **values**()
  - ✓ Returns all enumerated values for the type in an array

```
Day[] days = Day.values();
```

- ❑ You can use a regular for loop in (a) or foreach loop in (b) to process all the values in the array.

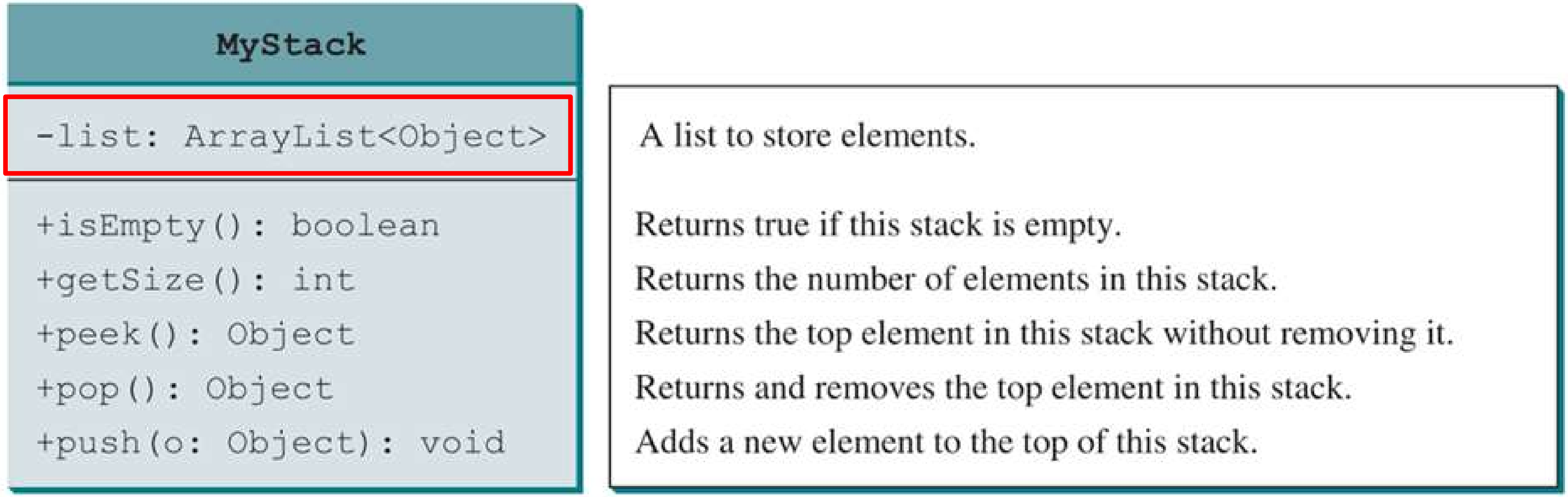


```
for (int i = 0; i < days.length; i++)
  System.out.println(days[i]);
```

```
for (Day day: days)
  System.out.println(day);
```

Practice: **Composition**

1. Implement **MyStack**, a stack class to store objects, using **composition**. Please use ArrayList to implement MyStack as shown below.



MyStack	
-list: ArrayList<Object>	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack without removing it.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): void	Adds a new element to the top of this stack.

Practice: **Inheritance**

2. Implement **MyStackInheritance**, a new stack class that **extends ArrayList**.

3. Write a **test file** for both the **MyStack** class and the **MyStackInheritance** class.