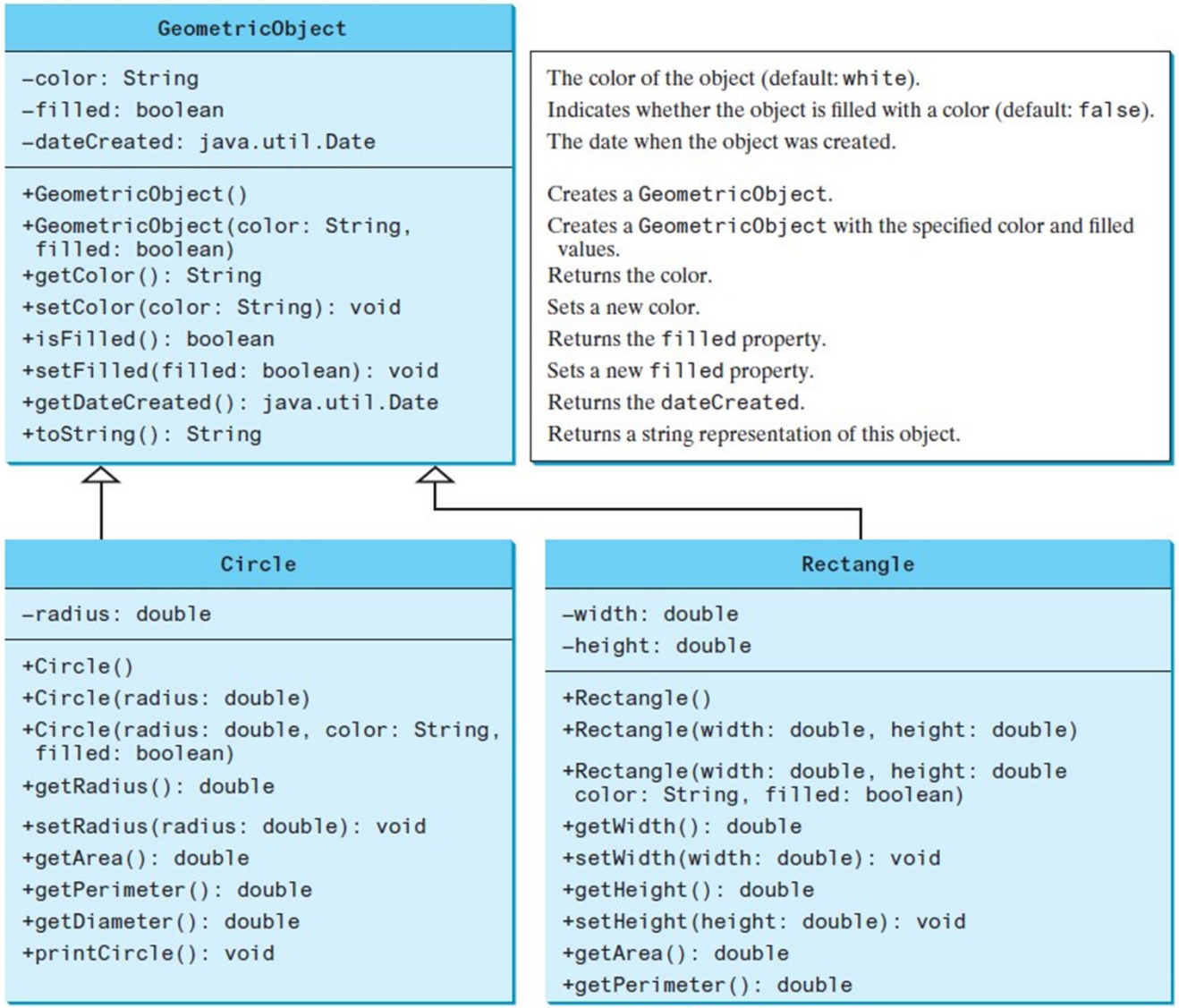
# Abstract Classes

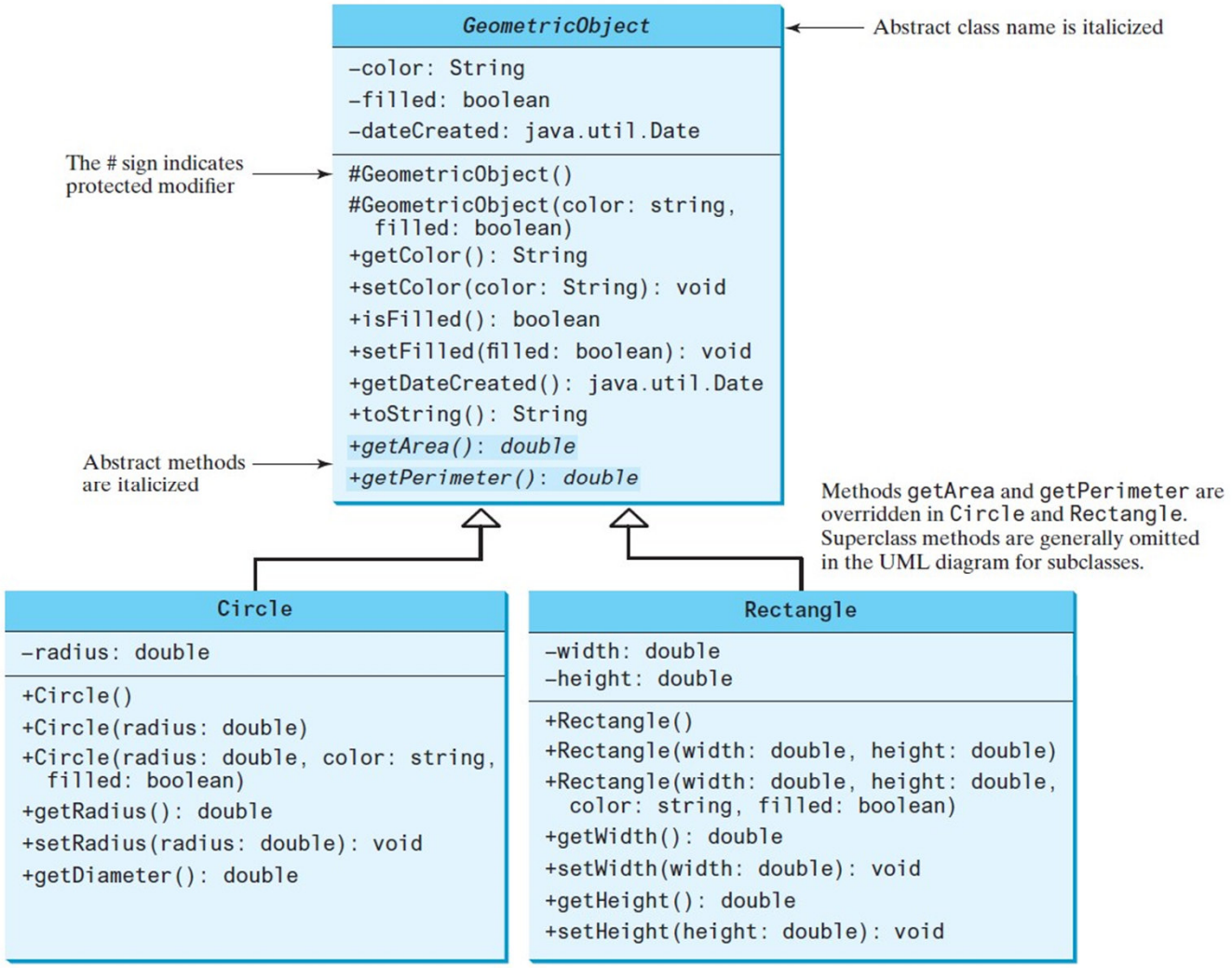
# Introduction

- Class design should ensure a superclass contains common features of its subclasses.
- If you move from a subclass back up to a superclass, the classes become more general and less specific.
- Sometimes, a superclass is so abstract it cannot be used to create any specific instances. Such a class is referred to as an *abstract class*.


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
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.
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

# Common Methods without Common Implementation

- Both Circle and Rectangle contain the getArea() and getPerimeter() methods
- The implementations of these common methods are completely different in different classes.
- There is no common implementation to be put into the superclass.
- The implementation completely depends on the specific type of GeometricObject
- Such methods are referred as *abstract methods*. They have signatures in the superclass, but no implementation.
- If a class contains abstract methods, it must declared as an *abstract class*.


GeometricObject
Abstract class name is italicized
-color: String
-filled: boolean
-dateCreated: java.util.Date
The # sign indicates protected modifier
#GeometricObject()
#GeometricObject(color: string, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
Abstract methods are italicized
+getArea(): double
+getPerimeter(): double
Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
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

```java
public abstract class GeometricObject {

  /** member variables */
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** Construct a default geometric object */
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }

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


.........

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();

}
```

```java
public class Rectangle extends GeometricObject {
  ……
  /** Return area */
  public double getArea() {
    return width * height;
  }

  /** Return perimeter */
  public double getPerimeter() {
    return 2 * (width + height);
  }

}
```

```java
public class Circle extends GeometricObject {
  ……
  /** Return area */
  public double getArea() {
    return radius * radius * Math.PI;
  }

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

}
```

# Abstract Classes and Abstract Methods

- Abstract classes are like regular classes, but you cannot create instances of abstract classes using the `new` operator.
- An abstract method is defined without implementation. Its implementation is provided by the subclasses.
- A class that contains abstract methods must be defined as abstract.

# Why Abstract Methods?

- You may be wondering what advantage is gained by defining the methods getArea() and getPerimeter() as abstract in the GeometricObject class.

```java
public class TestGeometricObject {
  /** Main method */
  public static void main(String[] args) {
  // Create two geometric objects
    GeometricObject geoObj1 = new Circle(5);
    GeometricObject geoObj2 = new Rectangle(5, 3);

    System.out.println("The two objects have the same area? "+
                       equalArea(geoObj1, geoObj2));

    // Display circle
    displayGeoObj(geoObj1);

    // Display retangle
   displayGeoObj(geoObj2);
  }

  /** Amethod for comparing the areas of two geometric objects */
  public static boolean equalArea(GeometricObject obj1, GeometricObject obj2) {
    return obj1.getArea()  == obj2.getArea();
  }

  /** A method for displaying a geometric object*/
  public static void displayGeoObj(GeometricObject obj) {
    System.out.println();
    System.out.println("The area is "+ obj.getArea() );
    System.out.println("The perimeter is "+ obj.getPerimeter());
  }
}
```

# Why Abstract Methods?

- Note you could not define the `equalArea` method for comparing whether two geometric objects have the same area if the `getArea` method were not defined in `GeometricObject`. Now you have seen the benefits of defining the abstract methods in `GeometricObject`.

# Abstract Method

- Abstract method capture the function, not the implementation.
- Abstract methods are placeholders that are meant to be overridden
- Thus, we don't have `private` or `static` abstract methods

# Abstract Method

- Abstract methods allow us to write code that makes use of a function without knowing the implementation.

```
public abstract class GeometricObject {

……

  /** Abstract method getArea */
  public abstract double getArea();

  public static  double  getTotalArea(GeometricObject[] ss) {
    double ta = 0.0;
    for (GeometricObject s : ss)
      ta += s.getArea();
    return ta;
  }
}
```

# Point to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

# Abstract method in abstract class

- An abstract method ***cannot*** be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented.

# Object cannot be created from abstract class

- An abstract class cannot be instantiated using the `new` operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of `GeometricObject` are invoked in the `Circle` class and the `Rectangle` class.

# abstract class without abstract method

- A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods. This abstract class is used as a base class for defining subclasses.

# concrete method overridden to be abstract

- A subclass can override a method from its superclass to define it as abstract. This is *very unusual*, but it is useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined as abstract.

# abstract class as type

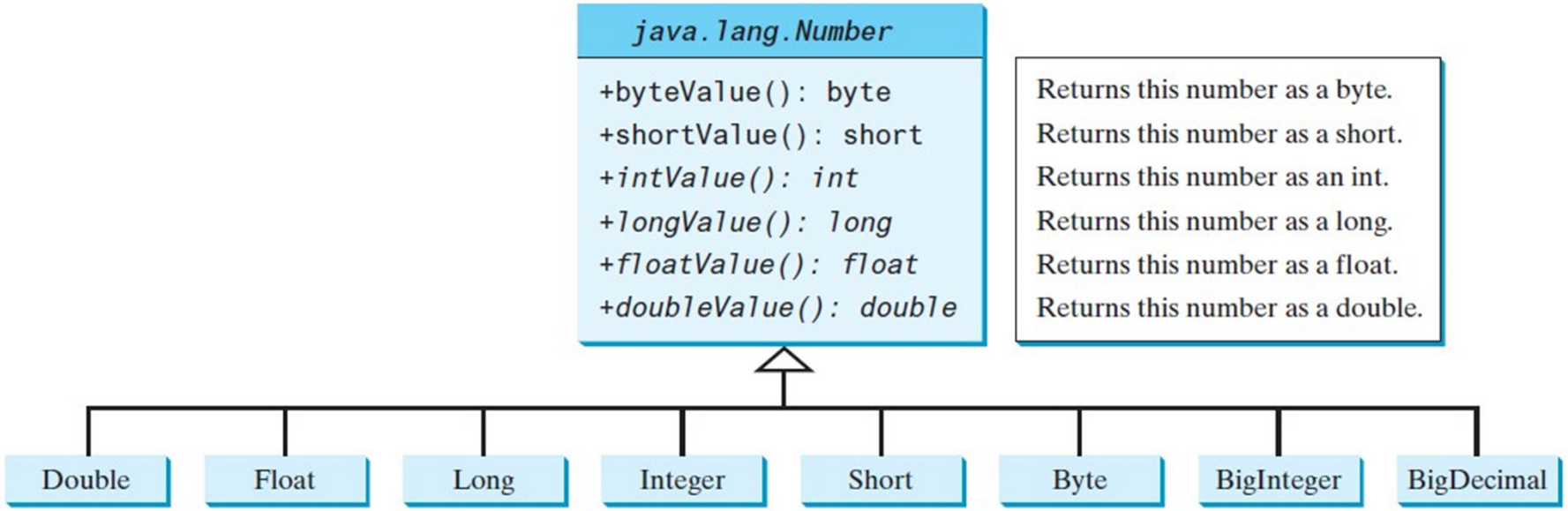
- You cannot create an instance from an abstract class using the **new** operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of the **GeometricObject** type, is correct:

```
GeometricObject[] objects = new GeometricObject[10];
```

- You can then create an instance of `GeometricObject` and assign its reference to the array like this:

```
objects[0] = new Circle();
```

# Case Study: the Abstract Number Class



# Largest Number Demo

LargestNumbers.java

```java
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.ArrayList;

public class LargestNumbers {

    public static void main(String[] args) {

        ArrayList<Number> list = new ArrayList<Number>();
        list.add(56);
        list.add(1556.32);

        list.add(new BigInteger("13416564562132313"));
        list.add(new BigDecimal("56.61643213654121331 12"));

        System.out.println("The largest number is " + getLargestNumber(list));

    }

    public static Number getLargestNumber(ArrayList<Number> list) {
        if (list == null || list.size() == 0)
            return null;
        Number number = list.get(0);
        for (int i = 1; i < list.size(); i++)
            if (number.doubleValue() < list.get(i).doubleValue())
                number = list.get(i);
        return number;
    }

}
```

RUN