Abstract Classes and Interfaces

Definitions

- An abstract class is a class that is declared abstract.

```
public abstract class GraphicObject {
   // declare fields
   // declare methods
}
```

- It may or may not include abstract methods.
- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

- Abstract classes cannot be instantiated, but they can be subclassed.

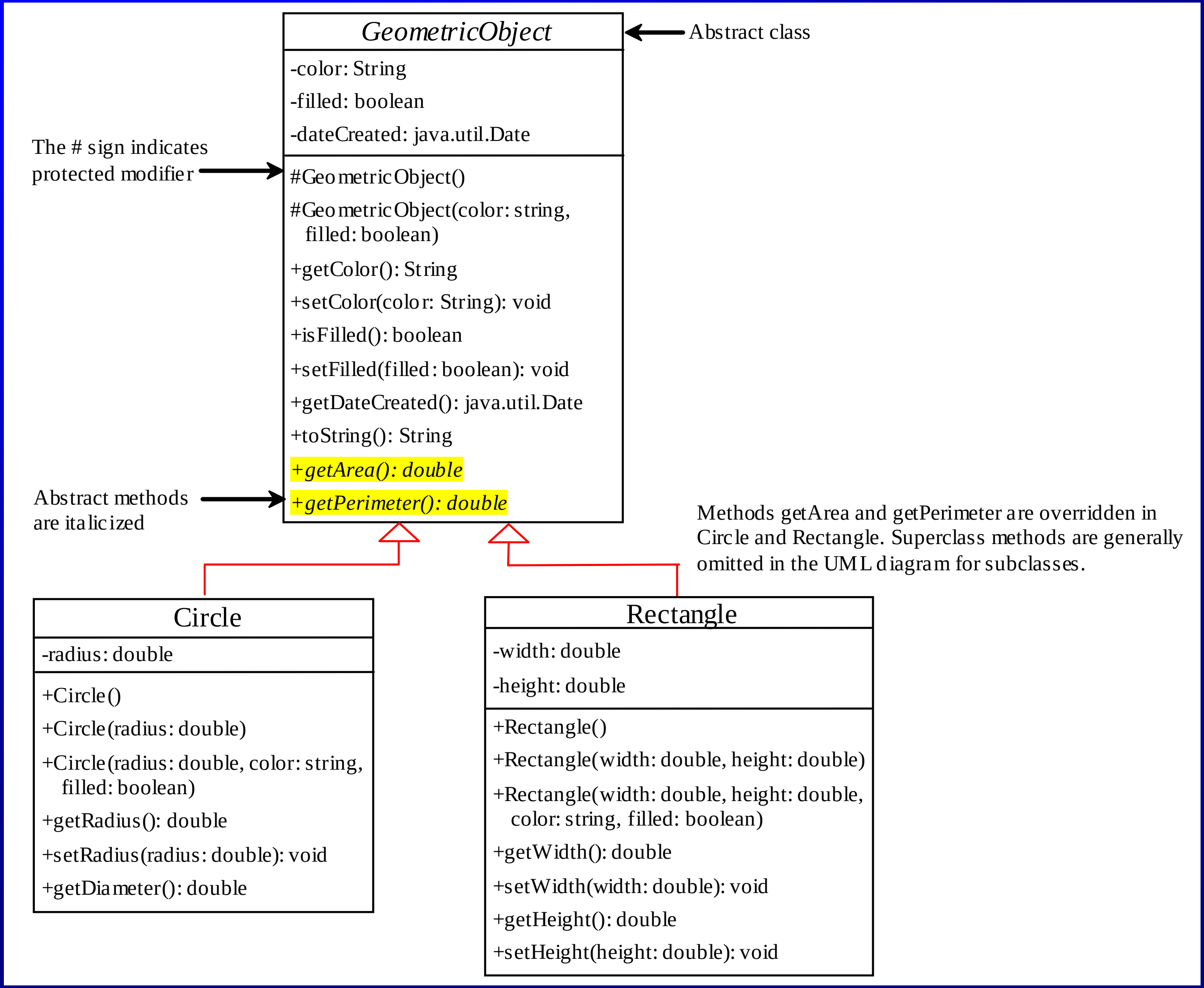
Definitions

- If a class includes abstract methods, then the class itself must be declared abstract, as in:

```
public abstract class GraphicObject {
   // declare fields
   // declare nonabstract methods
   abstract void draw();
}
```

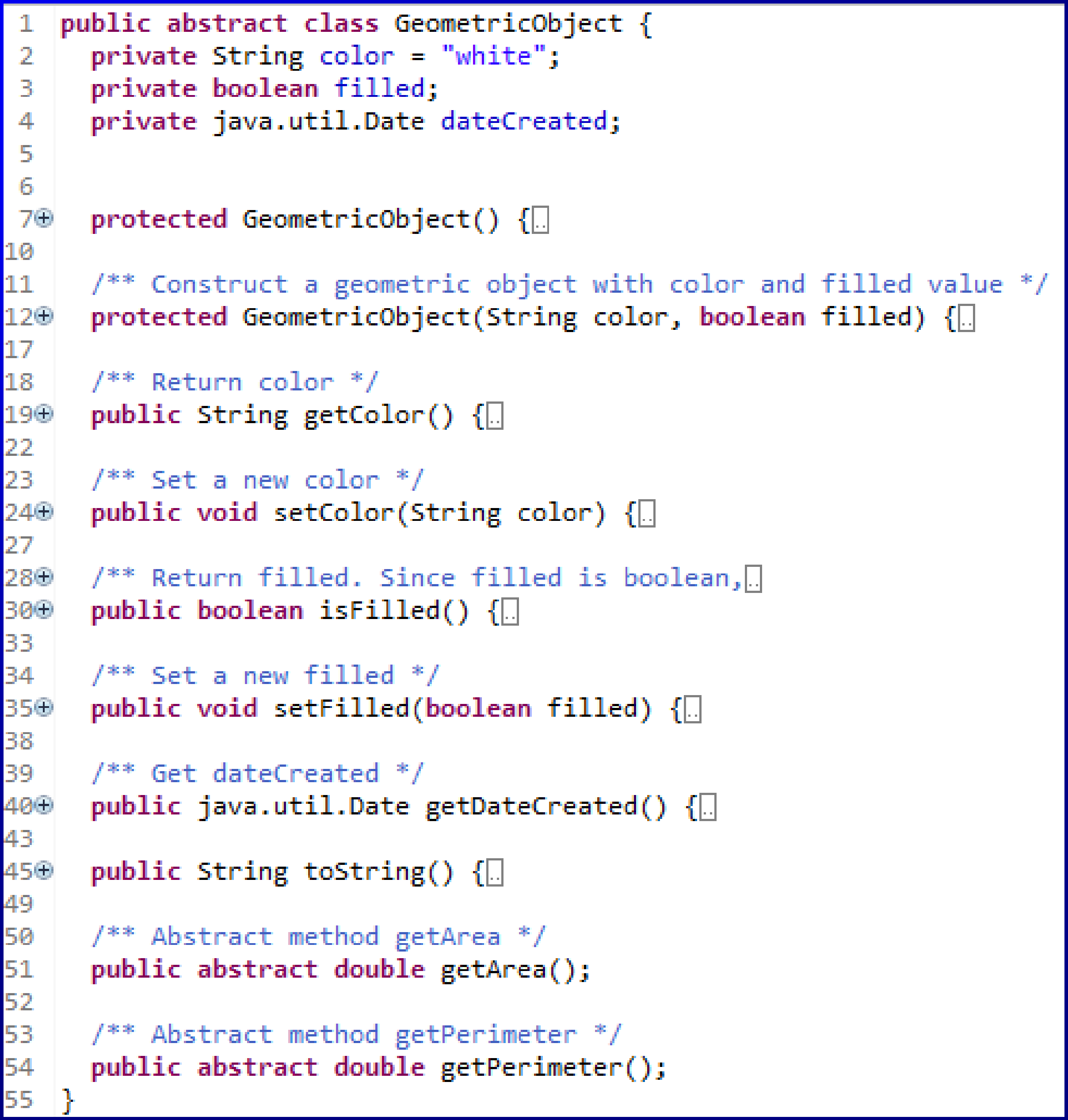
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class.
- However, if it does not, then the subclass must also be declared abstract.

Abstract Classes and Abstract Methods


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
+getArea(): double
+getPerimeter(): double
Abstract class
The # sign indicates protected modifier
Abstract methods are italicized
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

Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.

GeometricObject class



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


  protected GeometricObject() {

  /** Construct a geometric object with color and filled value */
  protected GeometricObject(String color, boolean filled) {

  /** Return color */
  public String getColor() {

  /** Set a new color */
  public void setColor(String color) {

  /** Return filled. Since filled is boolean,
  public boolean isFilled() {

  /** Set a new filled */
  public void setFilled(boolean filled) {

  /** Get dateCreated */
  public java.util.Date getDateCreated() {

  public String toString() {

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

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

Abstract Methods in Abstract classes

- An abstract method cannot be contained in a non-abstract class.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract. In other words, in a non-abstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.

Objects Cannot Be Created from Abstract Classes

- 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 Classes Without Abstract Method

- A class that contains abstract methods must be abstract.
- However, it is possible to define an abstract class that contains no abstract methods. In this case, you still cannot create instances of the class using the new operator.
- This class is only used as a base class for defining a new subclass.

Superclass of Abstract Class May be Concrete

- A subclass can be abstract even if its superclass is concrete.
- For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

Concrete Method Overridden to be Abstract

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

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

Interfaces

What is an interface?
Why is an interface useful?

- An interface is a class like construct that contains only constants and abstract methods.
- In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects.
- For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {
  constant declarations;
  method declarations;
}
```

Example:

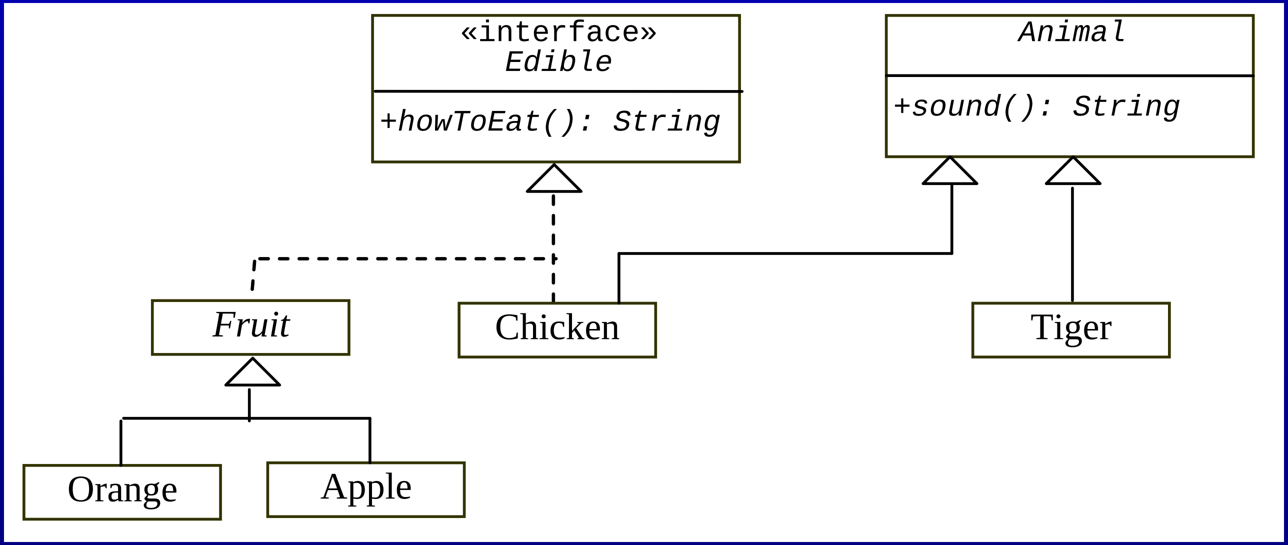
```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Interface is a Special Class

- An interface is treated like a special class in Java.
- Each interface is compiled into a separate bytecode file, just like a regular class.
- Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class.
- For example, you can use an interface as a data type for a variable, as the result of casting, and so on.

Example

We can use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface.



```java
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

```java
abstract class Animal {
  /** Return animal sound */
  public abstract String sound();
}
```

```java
class Tiger extends Animal {
  @Override
  public String sound() {
    return "Tiger: RROOAARR";
  }
}
```

```java
abstract class Fruit implements Edible {
  // Data fields, constructors, and methods omitted here
}
```

```java
class Chicken extends Animal implements Edible {
  @Override
  public String howToEat() {
    return "Chicken: Fry it";
  }

  @Override
  public String sound() {
    return "Chicken: cock-a-doodle-doo";
  }
}
```

```java
class Orange extends Fruit {
  @Override
  public String howToEat() {
    return "Orange: Make orange juice";
  }
}
```

```java
class Apple extends Fruit {
  @Override
  public String howToEat() {
    return "Apple: Make apple cider";
  }
}
```

Omitting Modifiers in Interfaces

All data fields are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:

```java
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```java
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

The Comparable Interface

- The Comparable interface defines the compareTo method for comparing objects.
- Suppose you want to design a generic method to find the larger of two objects of the same type, such as two students, two dates, two circles, two rectangles, or two squares. In order to accomplish this, the two objects must be comparable, so the common behavior for the objects must be comparable. Java provides the Comparable interface for this purpose.

The Comparable Interface

Definition:

```
// Interface for comparing objects, defined in java.lang
package java.lang;

public interface Comparable<E> {
    public int compareTo(E o);
}
```

- The compareTo method determines the order of this object with the specified object o and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than o.
- The Comparable interface is a generic interface. The generic type E is replaced by a concrete type when implementing this interface.

The Comparable Interface

Integer and BigInteger Classes

```java
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```java
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

String and Date Classes

```java
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```java
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

The Comparable Interface

Thus, numbers are comparable, strings are comparable, and so are dates. You can use the compareTo method to compare two numbers, two strings, and two dates. For example, the following code:

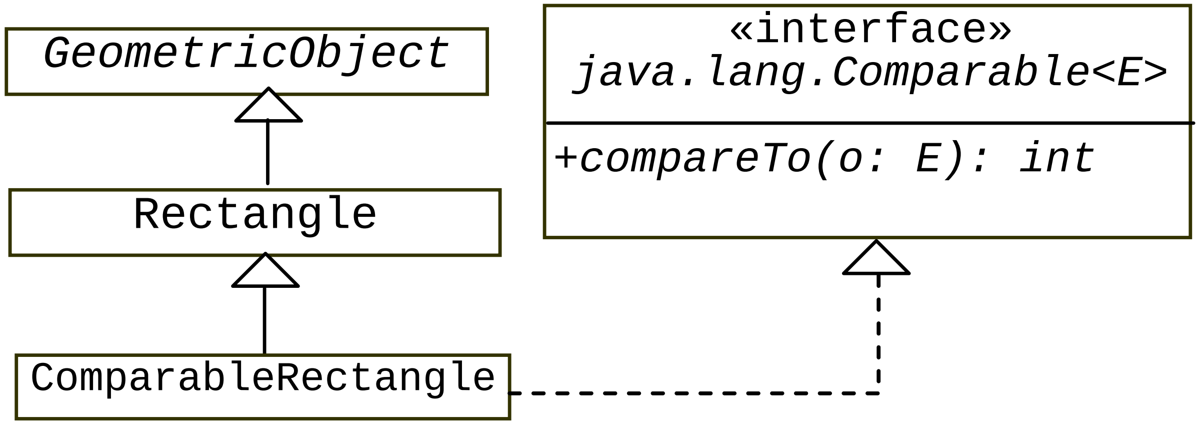
```
System.out.println(new Integer(3).compareTo(new Integer(5)));
System.out.println("ABC".compareTo("ABE"));
java.util.Date date1 = new java.util.Date(2013, 1, 1);
java.util.Date date2 = new java.util.Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));
```

displays

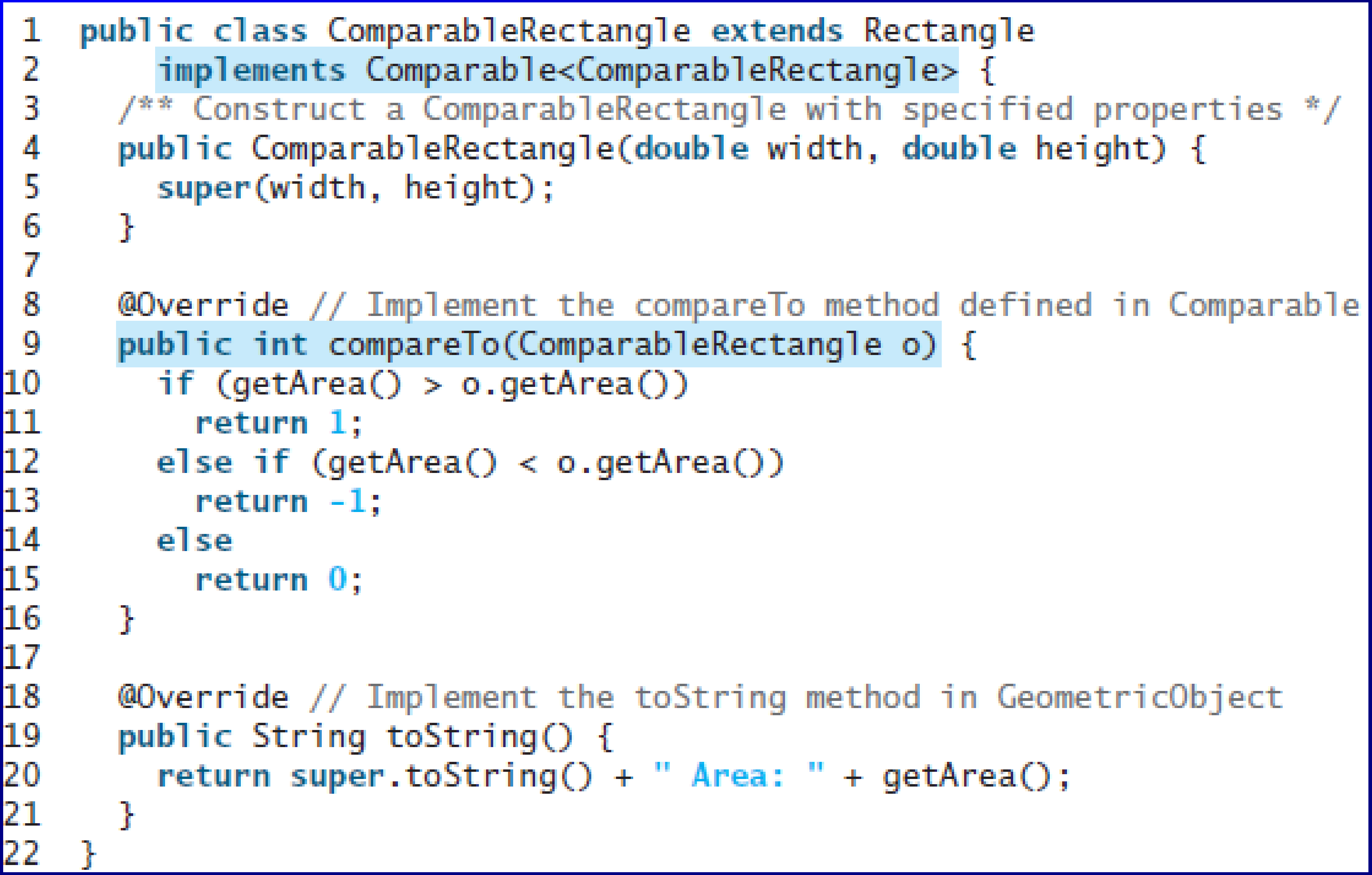
```
-1
-2
1
```

Example: Implement Comparable Rectangle

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.



Example: Implement Comparable Rectangle



```java
public class ComparableRectangle extends Rectangle
    implements Comparable<ComparableRectangle> {
  /** Construct a ComparableRectangle with specified properties */
  public ComparableRectangle(double width, double height) {
    super(width, height);
  }

  @Override // Implement the compareTo method defined in Comparable
  public int compareTo(ComparableRectangle o) {
    if (getArea() > o.getArea())
      return 1;
    else if (getArea() < o.getArea())
      return -1;
    else
      return 0;
  }

  @Override // Implement the toString method in GeometricObject
  public String toString() {
    return super.toString() + " Area: " + getArea();
  }
}
```

The `Cloneable` Interfaces

Marker Interface: An empty interface.

A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable properties. A class that implements the Cloneable interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

```
package java.lang;
public interface Cloneable {
}
```

Examples

Many classes (e.g., Date and Calendar) in the Java library implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code

```
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar)calendar.clone();
System.out.println("calendar == calendarCopy is " +
  (calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " +
  calendar.equals(calendarCopy));
```

displays

```
calendar == calendarCopy is false
calendar.equals(calendarCopy) is true
```

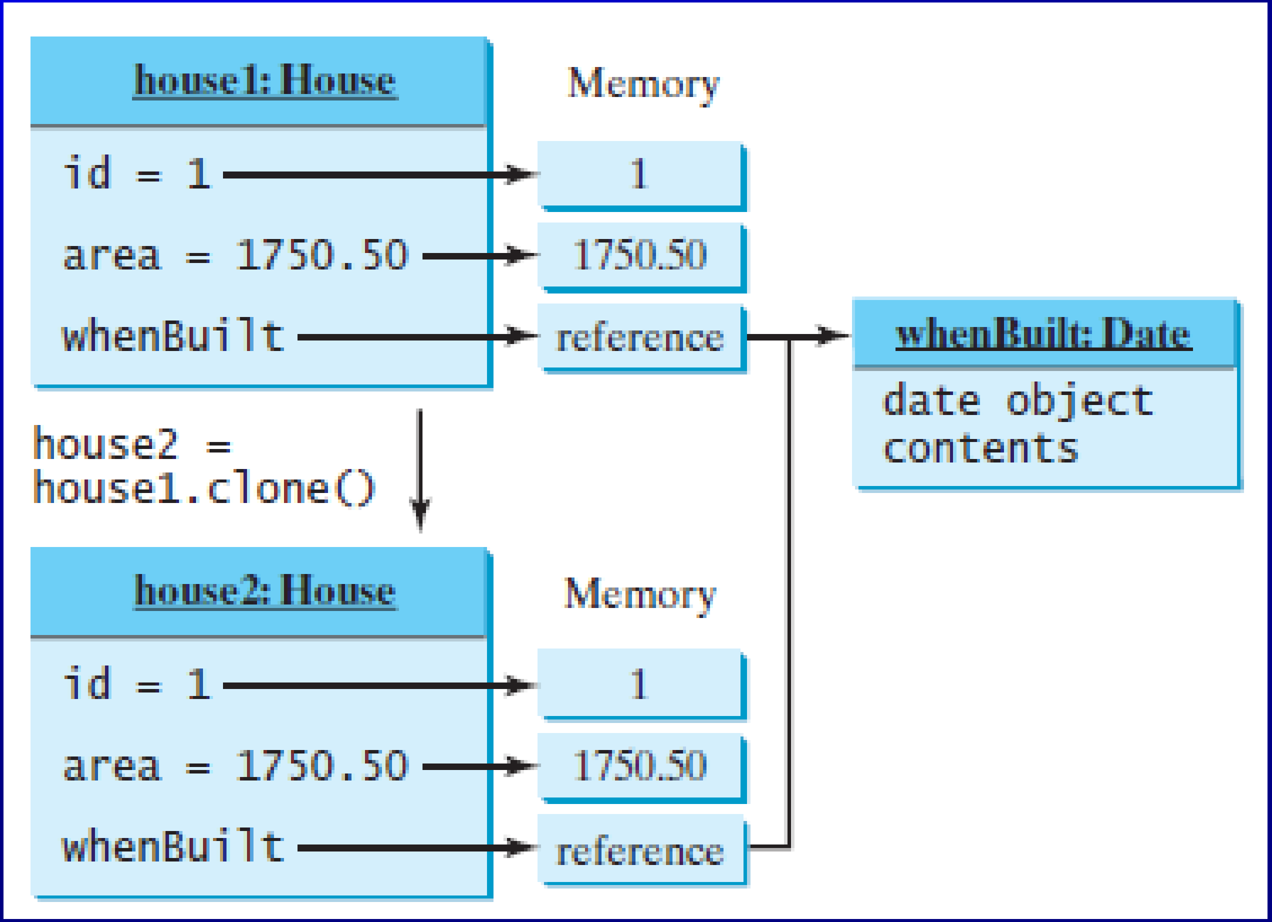
Implementing Cloneable Interface

- To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class.
- Warning a default clone method (super.clone()) might produce a shallow copy of an object. It might be necessary to create a custom clone method to perform a deep copy.

Shallow vs. Deep Copy

Shallow Copy (using default clone() method)

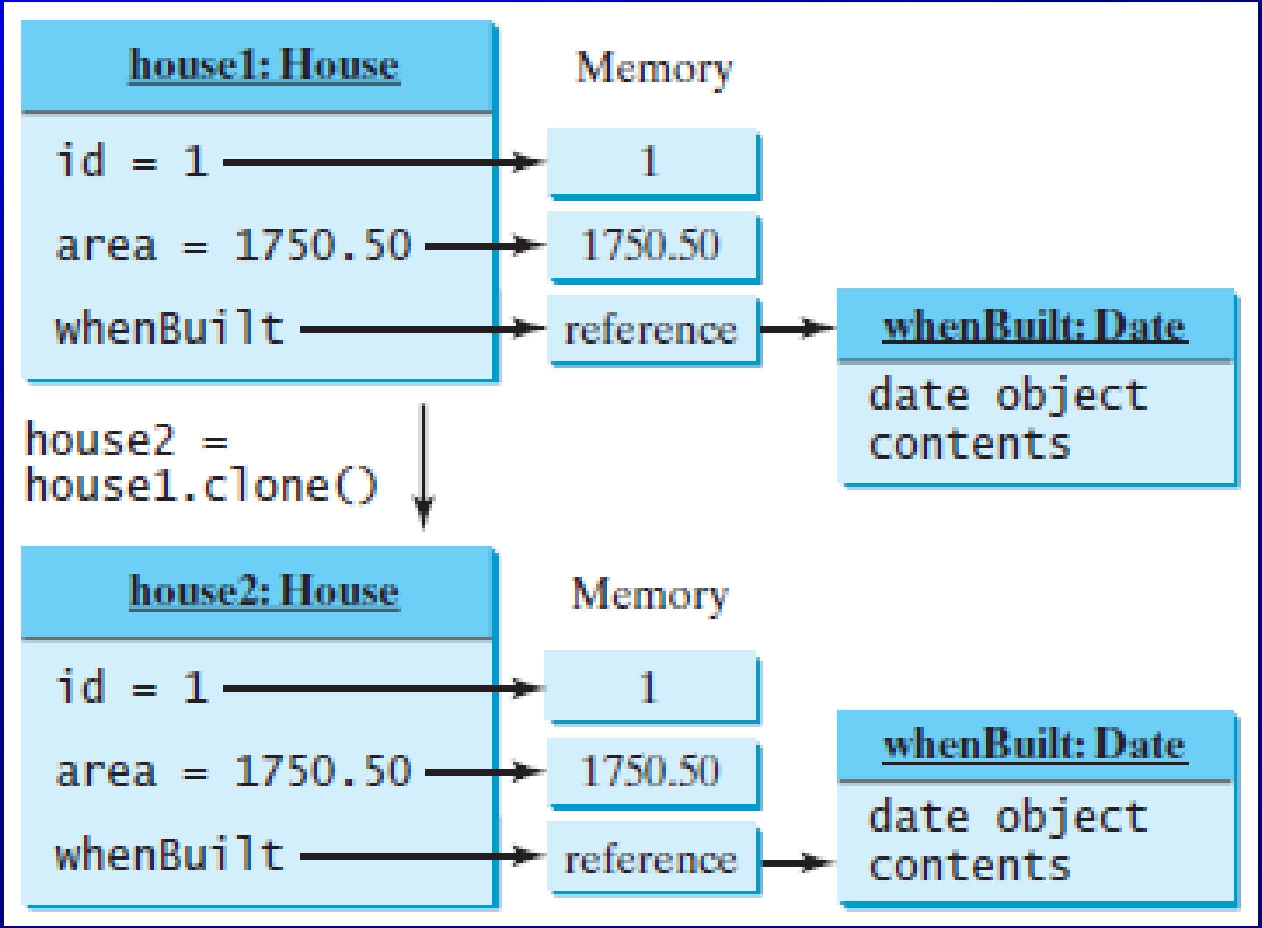
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();



Shallow vs. Deep Copy

Deep Copy (using a custom clone() method)

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```



Caution: conflict interfaces

- In rare occasions, a class may implement two interfaces with conflict information (e.g., two same constants with different values or two methods with same signature but different return type).
- This type of errors will be detected by the compiler.

How do you decide whether to use an interface or a class?

- In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person. So their relationship should be modeled using class inheritance.

- A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface.

- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface.

Class Design Guidelines

Class Design Guidelines
Cohesion

- A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff are different entities.
- A single entity with many responsibilities can be broken into several classes to separate the responsibilities. The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities.

Class Design Guidelines
Consistency

- Follow standard Java programming style and naming conventions.
- Choose informative names for classes, data fields, and methods. A popular style is to place the data declaration before the constructor and place constructors before methods.
- Make the names consistent. It is not a good practice to choose different names for similar operations. For example, the **length()** method returns the size of a **String**, a **StringBuilder**, and a **StringBuffer**. It would be inconsistent if different names were used for this method in these classes.
- In general, you should consistently provide a public no-arg constructor for constructing a default instance. If a class does not support a no-arg constructor, document the reason. If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.

Class Design Guidelines
Encapsulation

- A class should use the **private** modifier to hide its data from direct access by clients.
- This makes the class easy to maintain.
- Provide a getter method only if you want the data field to be readable.
- Provide a setter method only if you want the data field to be updateable.

Class Design Guidelines
Clarity

- Cohesion, consistency, and encapsulation are good guidelines for achieving design clarity.
- Additionally, a class should have a clear contract that is easy to explain and easy to understand.
- Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on how or when the user can use it, design the properties in a way that lets the user set them in any order and with any combination of values, and design methods that function independently of their order of occurrence.

Class Design Guidelines
Clarity

- Methods should be defined intuitively without causing confusion. For example, the **substring(int beginIndex, int endIndex)** method in the **String** class is somewhat confusing.
- The method returns a substring from **beginIndex** to **endIndex – 1**, rather than to **endIndex**. It would be more intuitive to return a substring from **beginIndex** to **endIndex**.
- You should not declare a data field that can be derived from other data fields.

Class Design Guidelines
Completeness

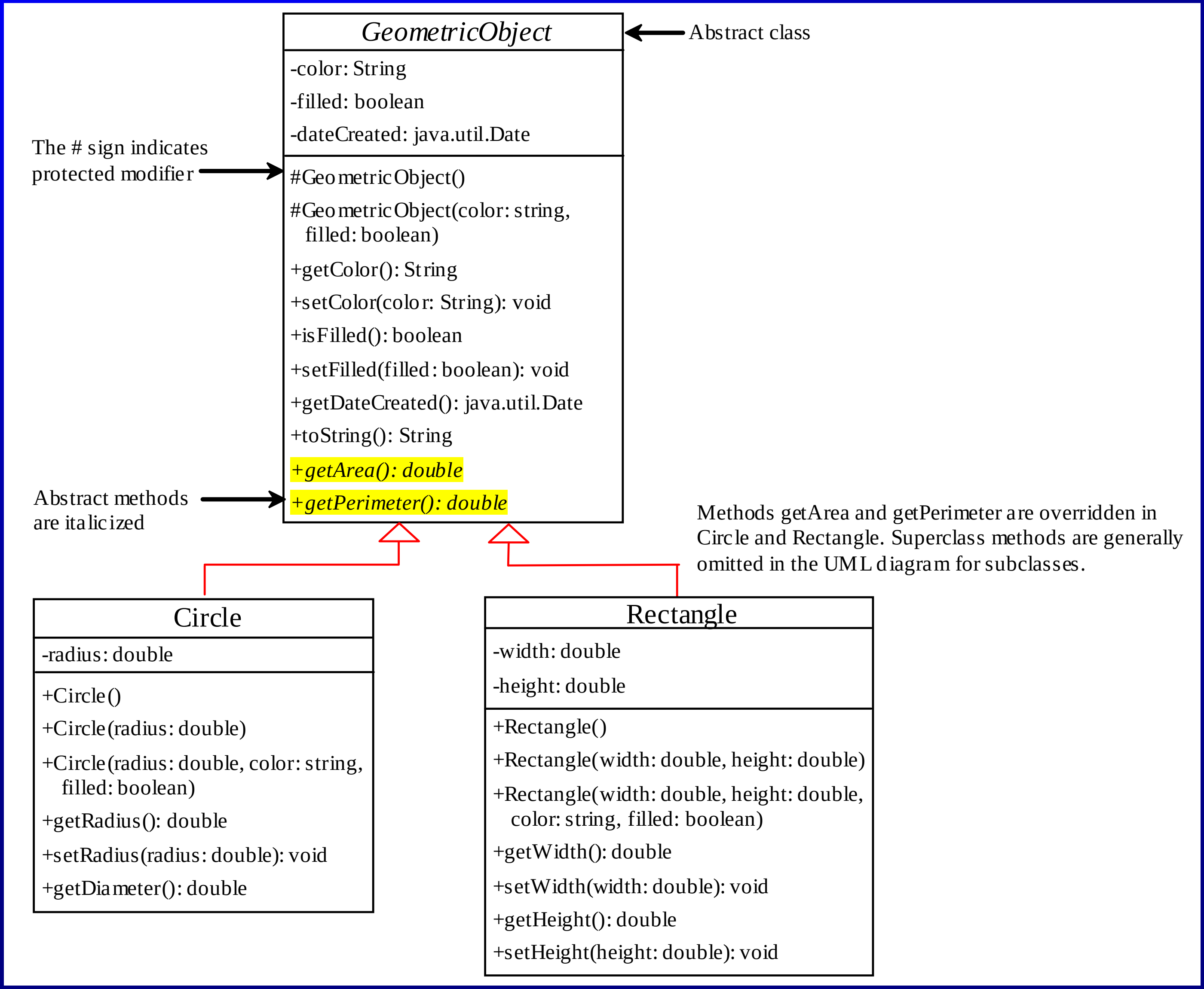
- Classes are designed for use by many different customers.
- In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through properties and methods.
- For example, the **String** class contains more than 40 methods that are useful for a variety of applications.

Class Design Guidelines
Instance vs. Static

- A variable or method that is dependent on a specific instance of the class must be an instance variable or method.
- A variable that is shared by all the instances of a class should be declared static.
- A method that is not dependent on a specific instance should be defined as a static method.
- Always reference static variables and methods from a class name (rather than a reference variable) to improve readability and avoid errors.
- Do not pass a parameter from a constructor to initialize a static data field. It is better to use a setter method to change the static data field.

Programming Challenge

Implement the classes below. Make Circle and Rectangle Comparable.


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
+getArea(): double
+getPerimeter(): double
Abstract class
The # sign indicates protected modifier
Abstract methods are italicized
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