Chapter 5

Implementing Inheritance

5.1 What is Inheritance?

Inheritance is a relation. It is also known as the *is-a* relation. It is a relation on classes. The relation captures that one class extends another class. The class that is extended is known as the *superclass* and the class that extends the superclass is known as the *subclass*. The subclass inherits certain features from the superclass. In particular, the subclass inherits the public non-static methods of its superclass.[1]

A golden rectangle is a rectangle made out of gold. Besides a width and a height, it also has a weight. The `GoldenRectangle` class extends the `Rectangle` class. It defines a special type of rectangle and adds a weight to each rectangle. Its API can be can be found at this link.

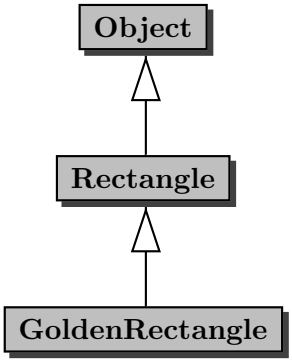
In the code snippet below, we create a `GoldenRectangle`. The `GoldenRectangle` class inherits the `scale` method of the `Rectangle` class. Phrased differently, since a `GoldenRectangle` is a `Rectangle`, we can `scale` it.

```
1 final int WIDTH = 3;
2 final int HEIGTH = 4;
3 final int WEIGHT = 80;
4 GoldenRectangle rectangle = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
5 final int FACTOR = 3;
6 rectangle.scale(FACTOR);
```

5.2 UML and Memory Diagrams

The inheritance relation can be depicted in UML class diagrams. For example, the diagram

[1] It also inherits public attributes of its superclass. However, we declare all non-static attributes to be private and we access all public static attributes and all public static methods via the class(name). Hence, we can restrict our attention to public non-static methods.



captures the inheritance relation between the classes mentioned in the previous section.

Recall that the state of an object consists of its non-static attributes and their values. Although the private non-static attributes of a class are not inherited by its subclasses, they are part of the state of instances of those subclasses. For example, the `Rectangle` class contains private non-static attributes named `width` and `height`. Since our `GoldenRectangle` class is a subclass of the `Rectangle` class, the attributes `width` and `height` and their values are part of the state of a `GoldenRectangle` object.

Since a `GoldenRectangle` has a weight, we will introduce an attribute named `weight` to capture that additional information.

Consider the client code of the previous section. When we reach the end of line 4, memory can be depicted as follows.

⋮	
100	`Client.main` invocation
`WIDTH`	3
`HEIGHT`	4
`WEIGHT`	80
`rectangle`	
`FACTOR`	
200	`GoldenRectangle` object
`width`	3
`height`	4
`weight`	80
⋮	

Note that we colour the part of the state which corresponds to attributes of the superclass. This reflects that these attributes cannot be accessed directly.

5.3 The `GoldenRectangle` Class

As we already mentioned in Section 5.1, a `GoldenRectangle` is a `Rectangle` made of gold and, hence, our class `GoldenRectangle` extends the `Rectangle` class. To reflect that the `GoldenRectangle` class is a subclass of the `Rectangle` class, we use the following class header.

```
1 public class GoldenRectangle extends Rectangle
```

5.3.1 The Attributes Section

A `GoldenRectangle` is a `Rectangle` with some additional information, namely its weight (in grams). Based on the signature of the three parameter constructor and the signature of the accessor `getWeight`, we decide to represent this additional information by means of an attribute of type `int`. We declare this attribute as follows.

```
1 private int weight;
```

Note that we do *not* introduce attributes to capture the width and height of the golden rectangle. This information is already represented by attributes of the `Rectangle` class and, hence, is already part of the state of a `GoldenRectangle` object.

As class invariant, we use

```
1 this.weight >= 0
```

5.3.2 The Constructors Section

Recall that the state of an object is initialized in the constructors. Since the attributes `width` and `height` are part of the state of a `GoldenRectangle` object, in the snippet

```
1 final int WIDTH = 3;
2 final int HEIGHT = 4;
3 final int WEIGHT = 80;
4 GoldenRectangle rectangle = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
```

the attributes `width`, `height` and `weight` get initialized. But how can we initialize the attributes `width` and `height`? These attributes are private and, hence, *cannot* be accessed directly. However, within the body of a constructor we can delegate to a constructor of the superclass. Hence, within the body of the three parameter constructor of the `GoldenRectangle`, we can delegate to the two parameter constructor of its superclass `Rectangle`.

To delegate to a constructor of the superclass, we use the keyword `super`. In the three parameter constructor of the `GoldenRectangle` class, we delegate to the two parameter constructor of the superclass `Rectangle` using `super(width, height)`. In this way, we initialize the attributes `width` and `height`. To initialize the weight, we use the private mutator `setWeight`. This results in the following constructor implementation.

```
1 public GoldenRectangle(int width, int height, int weight)
2 {
3    super(width, height);
4    this.setWeight(weight);
5 }
```

Consider the above lines of client code. When executing line 4 of the client code, first a block of memory for the attributes of the `GoldenRectangle` class and its superclass `Rectangle` is allocated.

⋮	
100	`Client.main` invocation
`WIDTH`	3
`HEIGHT`	4
`WEIGHT`	80
`rectangle`	
200	`GoldenRectangle` object
`width`	
`height`	
`weight`	
⋮	

Next, the three parameter constructor of the `GoldenRectangle` class is invoked. The corresponding invocation block can be depicted as follows.

⋮	
800	`GoldenRectangle` invocation
`this`	200
`width`	3
`height`	4
`weight`	80
⋮	

Within the body of the three parameter constructor of the `GoldenRectangle` class, the two parameter constructor of the `Rectangle` class is invoked by means of `super(width, height)`. Recall that we implemented the two parameter constructor of the `Rectangle` class as follows.

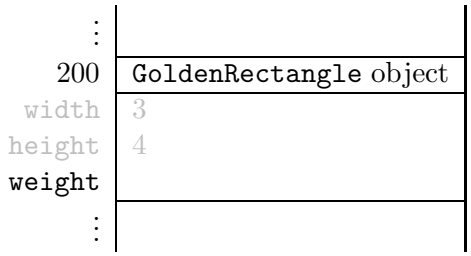
```
1 public Rectangle(int width, int height)
2 {
3    this.width = width;
4    this.height = height;
5 }
```

The invocation `super(width, height)` gives rise to an invocation block which can be depicted as follows.

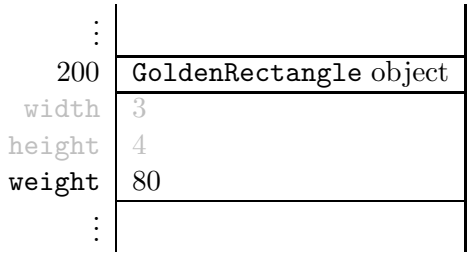
⋮	
900	`Rectangle` invocation
`this`	200
`width`	3
`height`	4
⋮	

Note that in the invocation `super(width, height)`, the parameter `this` is implicit.

Within the body of the two parameter constructor of the `Rectangle` class, we assign values to the attributes `width` and `height` of `this` object, that is, the object at address 200. Hence, once we reach the end of line 4 of the above constructor, the attributes `width` and `height` of the object at address 200 have the values 3 and 4, respectively.



Next, line 4 of the three parameter constructor of the `GoldenRectangle` class is executed. In this line, the `weight` attribute is initialized. Once the execution reaches the end of that line, the `GoldenRectangle` object can be depicted as follows.



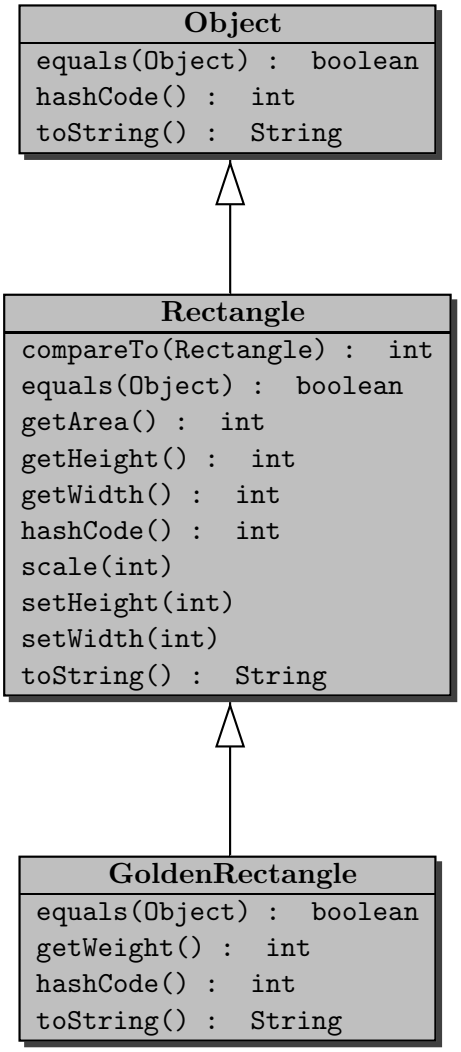
Next, we implement the copy constructor. It can be implemented in several different ways. For example, we can delegate to the copy constructor of the superclass.

```
1 public GoldenRectangle(GoldenRectangle copied)
2 {
3    super(copied);
4    this.setWeight(copied.getWeight());
5 }
```

The Java compiler requires that the `super(...)` statement be the first statement of the constructor.

5.3.3 The Methods Section

Before we implement the methods of our `GoldenRectangle` class, let us have a look at its superclasses `Object` and `Rectangle` and some of their methods.



Note that we override the methods `equals`, `hashCode` and `toString`. The method `getWeight` is new.

The `equals` Method

Two golden rectangles are the same if they have not only the same width and height, but also the same weight. To implement the `equals` method, we start with the usual skeleton.

```
1 public boolean equals(Object object)
2 {
3    boolean equal;
4    if (object != null && this.getClass() == object.getClass())
5    {
```

```
6        GoldenRectangle other = (GoldenRectangle) object;
7        equal = ???;
8     }
9     else
10    {
11       equal = false;
12    }
13 }
```

To check if `this` golden rectangle has the same weight as the `other` golden rectangle we can use

```
7 this.getWeight() == other.getWeight()
```

To check if both golden rectangles have the same width and height, we can delegate to the `equals` method of the `Rectangle` class as follows.[2]

```
7 super.equals(other)
```

Combing the two, we arrive at the following.

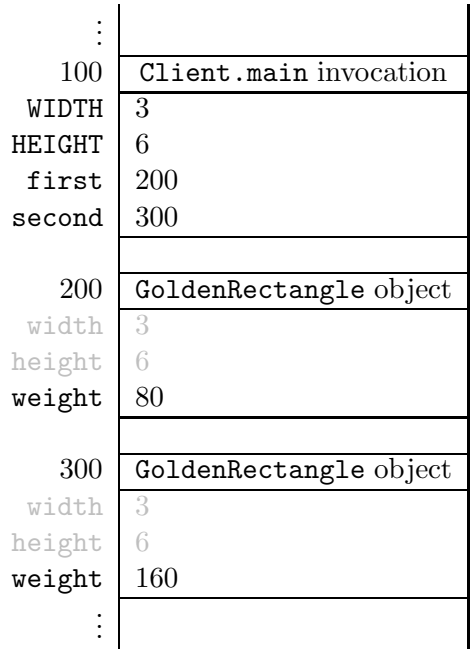
```
7 equal = super.equals(other) && this.getWeight() == other.getWeight();
```

Consider the following snippet of client code.

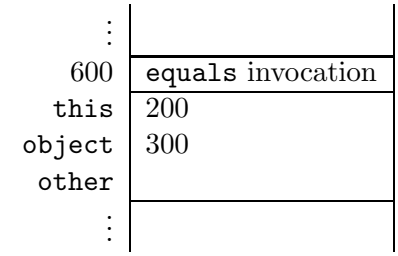
```
1 final int WIDTH = 3;
2 final int HEIGHT = 6;
3 final int WEIGHT = 80;
4 GoldenRectangle first = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
5 GoldenRectangle second = new GoldenRectangle(WIDTH, HEIGHT, 2 * WEIGHT);
6 output.println(first.equals(second));
```

Once we reach the end of line 5, memory can be depicted as follows.

[2]Note that `super.` is used to invoke a method of the superclass, whereas `super(...)` is used to delegate to a constructor of the superclass.

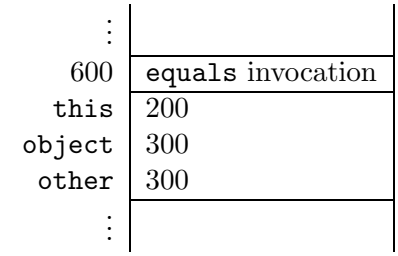


The invocation of the `equals` method in line 6 of the client code gives rise to the following invocation block.

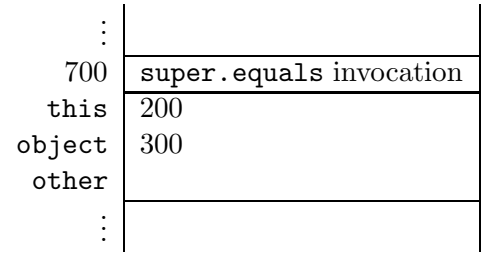


The invocation block contains the parameters `this` and `object` and the local variable `other`.

Now, let us consider the execution of the body of the `equals` method. Since `object` is different from `null` and both `this` and `object` refer to a `GoldenRectangle` object, line 6 and 7 of the `equals` method are executed. In line 6, the local variable `other` is initialized.



In line 7, the `equals` method of the superclass is invoked. This leads to the following invocation block.



Again, the invocation block contains the parameters `this` and `object` and the local variable `other`. Note that the parameter `this` is implicit in the invocation `super.equals(other)`. The `equals` method of the `Rectangle` class is invoked on the same object on which the `equals` method of the `ColouredRectangle` class is invoked.[3]

Let us consider the execution of the body of the `equals` method of the `Rectangle` class. Recall that we implemented the `equals` method in the `Rectangle` class as follows.

```
1  public boolean equals(Object object)
2  {
3     boolean equal;
4     if (object != null && this.getClass() == object.getClass())
5     {
6        Rectangle other = (Rectangle) object;
7        equal = (this.getWidth() == other.getWidth()) && (this.getHeight() ==
             other.getHeight());
8     }
9     else
10    {
11       equal = false;
12    }
13    return equal;
14 }
```

Since `object` is different from `null` and both `this` and `object` refer to a `GoldenRectangle` object, line 6 and 7 of the `equals` method are executed. In line 6, the local variable `other` is initialized to (the object at address) 300. In line 7, the width and height of `this` object, that is, the object at address 200, and of the `other` object, that is, the object at address 300, are compared.

In the next fragment of client code we compare a rectangle with a golden rectangle. Consider the following snippet of client code.

```
1  final int WIDTH = 3;
2  final int HEIGHT = 6;
3  final int WEIGHT = 80;
4  GoldenRectangle first = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
5  Rectangle second = new Rectangle(WIDTH, HEIGHT);
```

[3] One may think of `super.equals(other)` as `this.super.equals(other)` where `super.equals` denotes the `equals` method of the `superclass`. Note, however, that `this.super.equals(other)` is not valid Java syntax.

```
6 output.println(first.equals(second));
7 output.println(second.equals(first));
```

Neither `first` is equal to `second`, nor `second` is equal to `first`, since `first` and `second` are instances of different classes and, hence, their `Class` objects (returned by the accessor `getClass`) are different. We leave it to the interested reader to verify that the `equals` method satisfies the usual properties. We will come back to the `equals` method and its properties in Section 5.7.

The `hashCode` Method

The `hashCode` method returns an integer. This integer can be thought of as an abstraction of the state of the object. As a consequence, the value returned by the `hashCode` method is usually defined in terms of the values of the attributes of the class and its superclasses. For our `GoldenRectangle` class, the attributes `width`, `height` and `weight` and their values make up the state. The `hashCode` method of the superclass `Rectangle` already takes the attributes `width` and `height` into account. In the `hashCode` method of the `GoldenRectangle` class, we combine the result of the `hashCode` method of the `Rectangle` class and the `weight` attribute. The two integers can be combined in many ways. In superclass `Rectangle` we simply add the width and the height. Here we combine them as follows.[4]

```
1 public int hashCode()
2 {
3    final int BASE = 37;
4    return super.hashCode() + BASE * this.getWeight();
5 }
```

Recall that the `hashCode` method and the `equals` method are closely related. Since we override the `equals` method and the `hashCode` method, we have to check if they satisfy the usual property: if `x.equals(y)` returns true then `x.hashCode()` and `y.hashCode()` return the same integer for all `x` and `y` different from null. We leave it to the interested reader to check that this is indeed the case.

The `toString` Method

The `toString` method returns a string representation of the golden rectangle. For example, the snippet of client code

```
1 final int WIDTH = 3;
2 final int HEIGHT = 6;
3 final int WEIGHT = 80;
```

[4]Let b be a natural number. We call b the *base*. The *polynomial* hashcode of the integers $\langle i_0, \ldots, i_n \rangle$ is defined by

$$\sum_{k=0}^{n} i_k b^k.$$

The base b is usually a prime number. The details are beyond the scope of this book.

```
4 GoldenRectangle rectangle = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
5 output.println(rectangle);
```

produces the output

```
GoldenRectangle of width 3 and height 6 and weight 80
```

Note that part of this string, namely `Rectangle of width 3 and height 6`, is the same as the string representation returned by the `toString` method of the `Rectangle` class for a rectangle of width 3 and height 6. Hence, we can delegate to the `toString` method of the superclass to obtain this part. Therefore, we can implement the `toString` method as follows.

```
1 public String toString()
2 {
3    return "Golden" + super.toString() + " and weight " + this.getWeight();
4 }
```

The code of the `GoldenRectangle` class can be found by following this link.

5.4 The `PricingException` Class

Next we add the method

```
public double getPrice() throws PricingException
```

to our `GoldenRectangle` class. This method returns the price (in Canadian dollars) of the golden rectangle. The price is based on the weight of the golden rectangle and the current gold price. The method throws a `PricingException` if the gold price cannot be determined.

To determine the gold price, we utilize the method

```
public static double priceIn(String currencyCode)
  throws MalformedURLException, IOException, IndexOutOfBoundsException
```

of the `LondonGoldExchange` class. The API of this class can be found by following this link. The method returns the current price of one gram of gold in the given currency at the London Gold Exchange. The currency code for Canadian dollar is CAD. The method may throw three different types of exceptions. We refer the reader to the API of the `LondonGoldExchange` class for the conditions under which these exceptions are thrown. Since we want to hide these details from the client of our `getPrice` method, we introduce a new exception class, named `PricingException` which will be thrown whenever something goes wrong when determining the gold price.

First, we will implement the `getPrice` method. After that, we will implement the `PricingException` class.

The price of the golden rectangle is simply the product of its weight and the price of one gram of gold.

```
1 return this.getWeight() * LondonGoldExchange.priceIn("CAD");
```

Note that there is no need to introduce an attribute for the price, since it would be redundant. Also note that we should not cache the price, because the gold price varies over time.

Whenever the `priceIn` method throws an exception, our `getPrice` method throws a `PricingException`. This can be accomplished by catching the exceptions thrown by the `priceIn` method, and throwing instead a `PricingException`. This leads to the following implementation.

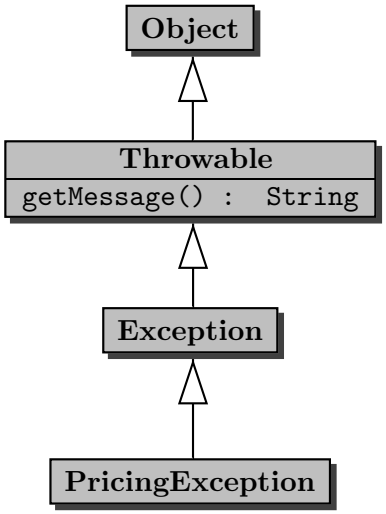
```
1  public double getPrice() throws PricingException
2  {
3     try
4     {
5        return this.getWeight() * LondonGoldExchange.getPrice("CAD");
6     }
7     catch (MalformedURLException e)
8     {
9        throw new PricingException("The gold price could not be determined");
10    }
11    catch (IOException e)
12    {
13       throw new PricingException("The gold price could not be determined");
14    }
15    catch (IndexOutOfBoundsException e)
16    {
17       throw new PricingException("The gold price could not be determined");
18    }
19 }
```

The `getPrice` method throws a `PricingException` if the gold price cannot be determined. Rather than throwing an instance of an already existing exception class, we decided to introduce a new exception class, `PricingException`, and to throw an instance this new class. Introducing our own exception class allows the client to easily detect that this exception is thrown by our `getPrice` method. Hence, this allows us to separate the ordinary code from the exception handling code.

Next, we implement the `PricingException` class. Its API can be can be found at this link. A `PricingException` is an exception and, hence, our class `PricingException` extends the class `Exception`. This is reflected in the header of our class.

```
1  public class PricingException extends Exception
```

Since the `PricingException` class extends the `Exception` class, it inherits all public non-static methods of the `Exception` class and its superclasses `Throwable` and `Object`. For example, it inherits the `getMessage` method of the `Throwable` class.



5.4.1 The Attributes Section

A `PricingException` is just a special type of exception, but contains no additional information. Therefore, we do not introduce any new attributes.

5.4.2 The Constructors Section

Recall that the private non-static attributes of the superclasses are part of the state. For example, the `Throwable` class contains a private non-static attribute named `detailMessage` of type `String`. This attribute contains the string which is returned by the `getMessage` method. Since our `PricingException` class is a subclass of the `Throwable` class, the attribute `detailMessage` and its value are part of the state of a `PricingException` object.

To initialize the attributes of the superclasses, such as `detailMessage`, we delegate to a constructor of the superclass.

```
1 public PricingException(String message)
2 {
3    super(message);
4 }
```

Note that the `Throwable` class does not contain a mutator `setMessage` and, hence, the attribute `detailMessage` can only be initialized by delegation to a constructor of the superclass `Exception`. That constructor of the superclass `Exception` in turn delegates to a constructor of its superclass, `Throwable`. In that constructor of the `Throwable` class the attribute `detailMessage` is initialized.

To implement the default constructor of our `PricingException` class, we simply delegate to the default constructor of its superclass.

```
1 public PricingException()
2 {
3    super();
4 }
```

5.4.3 The Methods Section

Our `PricingException` class contains no new public methods. Hence, we do not have to implement any. Note, however, that our class inherits methods from the classes `Object` and `Throwable` such as `equals` and `getMessage` (the class `Exception` contains no new public methods either).

The code of the `PricingException` class can be found by following this link.

5.5 The `ColouredRectangle` Class

Next, we consider a different extension of the `Rectangle` class. This time we add some colour to the rectangle. Its API can be can be found at this link. To reflect that our `ColouredRectangle` class extends the `Rectangle` class, we use the following class header.

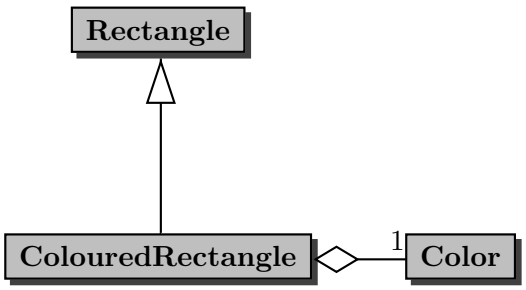
```
1 public class ColouredRectangle extends Rectangle
```

5.5.1 The Attributes Section

A `ColouredRectangle` is a `Rectangle` with some additional information, namely its colour. Based on the signature of the three parameter constructor and the signature of the accessor `getColour` and the return type of the mutator `setColour`, we decide to represent this additional information by means of the attribute of type `Color` (the class `Color` is part of the `java.awt` package). We declare the corresponding attribute as follows.

```
1 private Color colour;
```

The classes `ColouredRectangle`, `Rectangle` and `Color` are related as follows.



Note that this example combines inheritance and aggregation.

5.5.2 The Constructors Section

The API contains two constructors: a three parameter constructor and a two parameter constructor. Let us first consider the three parameter constructor. Again, we delegate to the superclass to initialize its part of the state.

```
1 public ColouredRectangle(int width, int height, Color colour)
2 {
```

```
3     super(width, height);
4     this.setColour(colour);
5  }
```

In the two parameter constructor, the width and height are set to the given `width` and `height`, and the colour is set to be white. In this case we can delegate to the three parameter constructor as follows.

```
1  public ColouredRectangle(int width, int height)
2  {
3     this(width, height, Color.WHITE);
4  }
```

5.5.3 The Methods Section

Of the methods of the `ColouredRectangle` class, we only discuss the `equals` method. Two coloured rectangles are the same if they not only have the same width and height, but also the same colour. To compare the width and the height, we delegate to the `equals` method of the superclass. This leads to the following implementation.

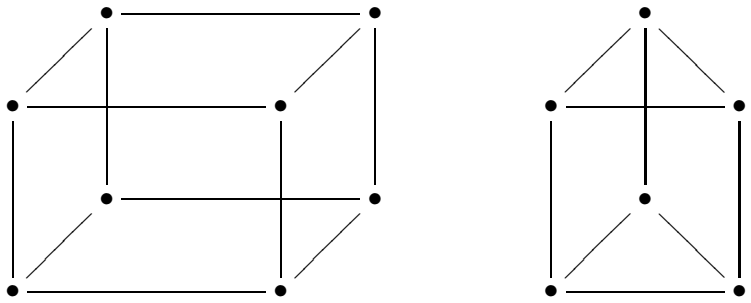
```
1  public boolean equals(Object object)
2  {
3     boolean equal;
4     if (object != null && this.getClass() == object.getClass())
5     {
6        ColouredRectangle other = (ColouredRectangle) object;
7        equal = super.equals(other) && this.getColour().equals(other.getColour()
             );
8     }
9     else
10    {
11       equal = false;
12    }
13    return equal;
14 }
```

Note that we use the `equals` method of the `Rectangle` class (through inheritance) and the `equals` method of the `Color` class (through aggregation).

The code of the `ColouredRectangle` class can be found by following this link.

5.6 Implementing Abstract Classes

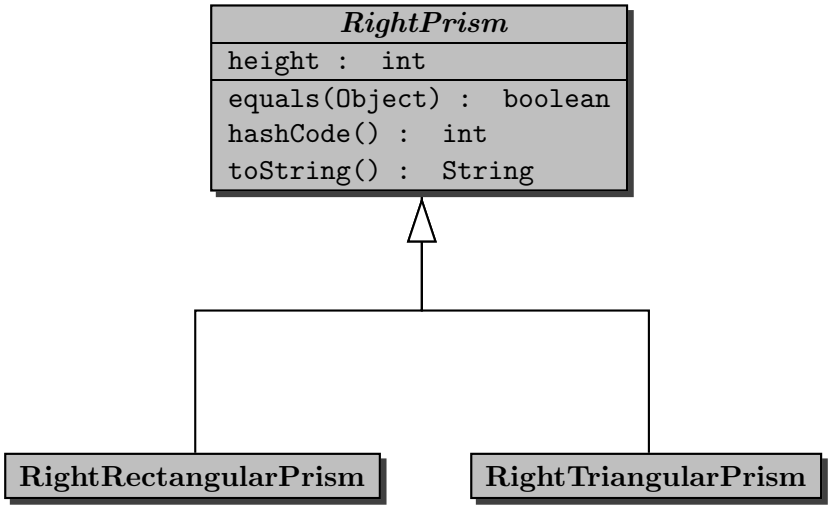
Abstract classes are often introduced to avoid code duplication. For example, consider the classes `RightRectangularPrism` and `RightTriangularPrism`.



The base of a right rectangular prism is a rectangle and the base of a right triangular prism is a triangle. In a right rectangular prism all angles are 90°. In a right triangular prism, apart from the six angles of the two triangles, all other angles are 90°. Both a right rectangular prism and a right triangular prism have a height. Hence, we may represent this information by the following attribute.

```
1 private int height;
```

Rather than duplicating this code in the `RightRectangularPrism` class and in the `RightTriangularPrism` class, we introduce a common superclass, named `RightPrism`, that contains the common code. Its API can be can be found at this link.



To prevent the client from creating an instance of the `RightPrism` class, we declare the class to be abstract.

```
1 public abstract class RightPrism
```

If the client were to attempt to create an instance of this abstract class

```
1 final int HEIGHT = 5;
2 RightPrism prism = new RightPrism(HEIGHT);
```

then a compile time error would occur

```
RightPrismClient.java:2: RightPrism is abstract; cannot be instantiated
        RightPrism prism = new RightPrism(HEIGHT);
                           ^
1 error
```

As we already mentioned, the `RightPrism` class contains the above declaration of the attribute `height`. The corresponding accessor and mutator are private.

Although the client cannot create an instance of the `RightPrism` class, we do add a constructor to the class so that subclasses can delegate to it to initialize the `height` attribute.

```
1  public RightPrism(int height)
2  {
3     super();
4     this.setHeight(height);
5  }
```

Note that we delegate to the default constructor of the superclass, the `Object` class. If a constructor does not explicitly invoke a superclass constructor, then the Java compiler automatically inserts `super()` and, hence, `super()` can be left out in the above constructor.[5]

The `RightPrism` class contains the methods `equals`, `hashCode` and `toString` which all depend on the `height` attribute. For example, the `equals` method can be implemented as follows.

```
1  public boolean equals(Object object)
2  {
3     boolean equal;
4     if (object != null && this.getClass() == object.getClass())
5     {
6        RightPrism other = (RightPrism) object;
7        equal = this.getHeight() == other.getHeight();
8     }
9     else
10    {
11       equal = false;
12    }
13 }
```

[5]If the Java compiler automatically inserts `super()` and the superclass does not have a default constructor, then a compile-time error will occur.

The code of the `RightPrism` class can be found by following this link.

Let us only have a look at the `RightRectangularPrism` class. Its API can be can be found at this link. This class extends the abstract `RightPrism` class which is reflected in the class header.

```
1 public class RightRectangularPrism extends RightPrism
```

As a consequence, the `RightRectangularPrism` class inherits methods from the `Object` class, such as `getClass`, and from the `RightPrism` class, namely `equals`, `hashCode`, and `toString`. Furthermore, the attribute `height`, although not inherited, is part of the state of a `RightRectangularPrism` object.

Because a right rectangular prism can be represented by its width, height and depth, and since the height is already represented in the superclass, we only need to introduce the following two attributes.

```
1 private int width;
2 private int depth;
```

The state of a `RightRectangularPrism` object, consisting of its `width`, `height` and `depth`, is initialized as follows.

```
1 public RightRectangularPrism(int width, int height, int depth)
2 {
3    super(height);
4    this.setWidth(width);
5    this.setDepth(depth);
6 }
```

Note that we delegate to the constructor of the `RightPrism` class.

In the `RightRectangularPrism` class we override the methods `equals`, `hashCode` and `toString`. For example, the `equals` method can be implemented as follows.

```
1  public boolean equals(Object object)
2  {
3     boolean equal;
4     if (object != null && this.getClass() == object.getClass())
5     {
6        RightRectangularPrism other = (RightRectangularPrism) object;
7        equal = super.equals(other)
8                && this.getWidth() == other.getWidth()
9                && this.getDepth() == other.getDepth();
10    }
11    else
12    {
13       equal = false;
14    }
15 }
```

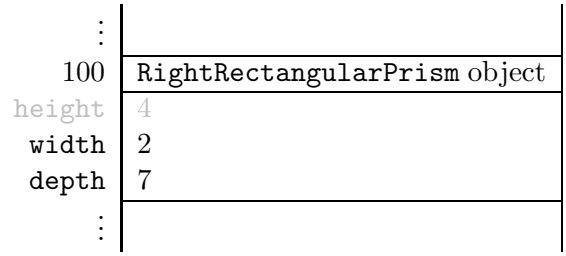
Note that we delegate to the `equals` method of the `RightPrism` class.

The code of the `RightRectangularPrism` class can be found by following this link.

5.7 Beyond the Basics

5.7.1 The `getVolume` Method

We want to add a `getVolume` method to the classes that represent right prisms such as `RightRectangularPrism` and `RightTriangularPrism`. Consider, for example, the `RightRectangularPrism` class. In this case, its volume is defined as the product of its height, width and depth. However, in the `RightRectangularPrism` we cannot compute this product since the attribute `height` is not accessible, as is reflected in the following object block.



Each right prism has a base area. However, the base area is computed differently for different types of right prisms. For example, for a right rectangular prism its base area is the product of its width and depth. Note that this product can be computed in the `RightRectangularPrism` class as follows.

```
1 public int getBaseArea()
2 {
3    return this.getWidth() * this.getDepth();
4 }
```

The facts that each right prism has a base area but it cannot be computed in the `RightPrism` class, can be reflected by adding the following abstract method to the `RightPrism` class.

```
1 public abstract int getBaseArea();
```

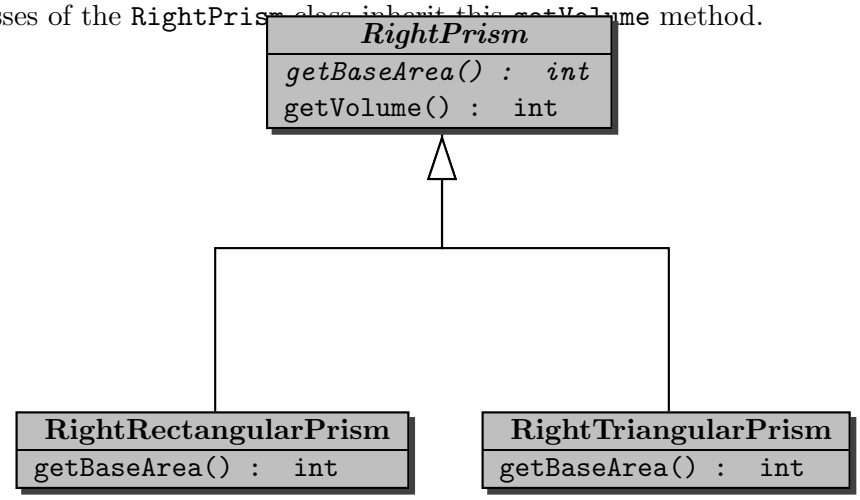
By introducing this method declaration, we enforce that each (non-abstract) subclass has to provide an implementation of the `getBaseArea` method. If we were to extend the `RightPrism` class and not implement the `getBaseArea` method, then we would get a compile time error such as

```
IncompleteRightPrism.java:1: IncompleteRightPrism is not abstract and
does not override abstract method getBaseArea() in RightPrism
public class IncompleteRightPrism extends RightPrism
       ^
1 error
```

Given that we have an (abstract) `getBaseArea` method in the `RightPrism` class, we can now implement the `getVolume` method in the `RightPrism` class as follows.

```
1  public int getVolume()
2  {
3     return this.getHeight() * this.getBaseArea();
4  }
```

All subclasses of the `RightPrism` class inherit this `getVolume` method.



When we compile the `RightPrism` class, the invocation `this.getBaseArea()` in the body of the `getVolume` method is bound to the `getBaseArea` method of the `RightPrism` class. At run time, the `getBaseArea` method is invoked on an instance of a (non-abstract) subclass of the `RightPrism` class. As we already mentioned above, this subclass has to implement the `getBaseArea` method. Hence, at run time the invocation `this.getBaseArea()` in the body of the `getVolume` method is bound to the `getBaseArea` method of that subclass of the `RightPrism` class.

5.7.2 Preconditions

Recall that we implemented the `scale` method of the `Rectangle` class as follows.

```
1  /**
2   * Scale this rectangle with the given factor.
3   *
4   * @param factor scaling factor.
5   * @pre. factor >= 0
6   */
7  public void scale(int factor)
8  {
9     this.width *= factor;
10    this.height *= factor;
11 }
```

The `Factory` class contains a method

```
public static Rectangle getInstance()
```

which returns a `Rectangle` with random width and height.

A client uses the above classes in an app as follows.

```
1 Rectangle rectangle = Factory.getInstance();
2 rectangle.scale(0);
3 output.println(rectangle);
```

When the client runs the app, it does not produce the expected output

```
Rectangle with width 0 and height 0
```

After inspecting the API of our `Rectangle` class, the client blames us since the client has satisfied the precondition of our `scale` method, yet our method does not return the expected string.

How is it possible that the `scale` method does not return the expected string? Are we to blame?

After inspecting our `scale` method, we are convinced that it is correct. Hence, we are not to blame. However, the client is not to blame either. So, who is to blame?

The implementer of the `Factory` class also wrote the `HiddenRectangle` class. This class extends the `Rectangle` class and overrides the `scale` method. The precondition of the `scale(factor)` method of the `HiddenRectangle` class is `factor` > 0 and the method does nothing in case the precondition is not met (recall that as an implementer we can do whatever we want if the precondition is not met).

The implementer of the `Factory` class implemented the `getInstance` method in such a way that it returns a `HiddenRectangle` object which is-a `Rectangle`. As a consequence, at compile time, the method call `rectangle.scale(0)` is bound to the `scale` method of our `Rectangle` class. However, at run time, the method call `rectangle.scale(0)` is bound to the `scale` method of the `HiddenRectangle` class. Hence, it does not set the width and height to zero.

Note that the precondition of the `scale` method in the `HiddenRectangle` class strengthens the precondition of the `scale` method in the `Rectangle` class: `factor > 0` is stronger than `factor >= 0` since the former implies the latter. As a consequence, the `Rectangle` class guarantees that the `scale` method works as expected if the argument 0 is provided, whereas the `HiddenRectangle` class does not. However, a `HiddenRectangle` object is-a `Rectangle` and, hence, should behave like a `Rectangle`. In particular, its `scale` method should work as expected if the argument 0 is provided. Therefore, we blame the implementer of the `HiddenRectangle` class: the precondition of the `scale` method should not have been strengthened.

In a subclass, we can weaken the precondition as is shown in the following alternative implementation of the `HiddenRectangle` class.

```
1 public class HiddenRectangle extends Rectangle
2 {
3    /**
4     * Scale this rectangle with the given factor.
5     * If the factor is negative then its absolute value is used.
6     *
7     * @param factor scaling factor.
8     * @pre. true
9     */
```

```
10    public void scale(int factor)
11    {
12       super.scale(Math.abs(factor));
13    }
14 }
```

5.7.3 Postconditions

Similarly, it can be shown that the postcondition of a method cannot be weakened in a subclass. It can be strengthened though.

5.7.4 Exceptions

From the API of the `Rectangle` class we can conclude that the `scale` method does not throw an exception. Hence, in the snippet

```
1 public static void main(String[] args)
2 {
3    Rectangle rectangle = Factory.getInstance();
4    final int FACTOR = 3;
5    rectangle.scale(FACTOR);
6 }
```

we do not need to enclose the `scale` invocation in a try block. As we have seen above, the `getInstance` method may return an instance of a subclass of the `Rectangle` class. If the subclass overrides the `scale` method and the `getInstance` method returns an instance of the subclass, then the overridden `scale` method is invoked in line 5. As a consequence, the overridden `scale` method cannot throw an exception either.

If we were to override the `scale` method in the `HiddenRectangle` class and throw a `RectangleException` if the argument of the `scale` method were negative and we were to compile the `HiddenRectangle` class, then we would get the following error message.

```
HiddenRectangle.java:9: scale(int) in HiddenRectangle cannot override scale(int)
in Rectangle;
overridden method does not throw RectangleException
    public void scale(int factor) throws RectangleException
                ^
1 error
```

In summary, when we override a method, we cannot throw any exception that is not thrown by the method in the superclass.

As we have seen, the `getPrice` method of the `GoldenRectangle` class may throw an exception of type `PricingException`. If we override the `getPrice` method in a subclass of the `GoldenRectangle` class, then we can throw an exception provided that it is an instance of `PricingException` or one of its subclasses.

5.7.5 The `equals` Method Revisited

In our implementation of the `equals` method, we use the `getClass` method to check if the two objects are of the same type. Can `instanceof` be used instead? Assume we use

```
1   if (object != null && object instanceof ColouredRectangle)
```

instead of

```
1   if (object != null && this.getClass() == object.getClass())
```

in the `equals` method of the `ColouredRectangle` class. The `equals` method of the `Rectangle` class can be modified similarly. Now consider the following snippet of client code.

```
1 final int WIDTH = 3;
2 final int HEIGHT = 6;
3 ColouredRectangle first = new ColouredRectangle(WIDTH, HEIGHT, Color.RED);
4 Rectangle second = new Rectangle(WIDTH, HEIGHT);
5 output.println(first.equals(second));
6 output.println(second.equals(first));
```

On the one hand, because `second` is not an instance of `ColouredRectangle`, the invocation `first.equals(second)` returns false. On the other hand, since `first instanceof Rectangle` returns true and `first` and `second` have the same width and height, the invocation `second.equals(first)` returns true. Hence, this implementation gives rise to an `equals` method that is not symmetric. Since symmetry is part of the postcondition of the `equals` method of the `Object` class and, as we have seen above, the postcondition cannot be weakened, the `equals` method of the `ColouredRectangle` class should be symmetric as well. Since the implementation of the `equals` method using `instanceof` weakens the postcondition, it is incorrect.