PART 4

We have now finished our introduction to the main concepts of object-oriented programming.
More specifically, we have a good idea of what an object is.

As far as we have learned, a typical class declaration looks like:

```
class SomeClass
{
        // fields ...

        // constructors ...

        // methods ...
}
```

Now, it is time to make the final touches !

Once again, some brainstorming would help.

Let's say your supervisor asked you to write a class that would represent Time. Here are the specifications that your supervisor has provided:

- It should be composed of the hour, minute and second.
- It should allow me to create a new Time object, when the hour, minute and second properties are provided.
- It must be able to print its time in a nice way.

As an important side note, he adds that he would use this Time class in his new application. Therefore, he asks you to be very careful while coding the Time class since it would play a critical role in his application.

After accepting this critical task, you start working on the class. You do your real best to come up with the bug-free implementation.

Reading the specifications, you start implementing the Time class. A draft would look like:

- int hour
- int minute
- int second
- A constructor that accepts the three parameters
- A method that prints the time

In correspondance with your draft, the implementation would look like:

```
class Time
{
        int hour;
        int minute;
        int second;

        Time(int initialHour, int initialMinute, int initialSecond)
        {
                hour = initialHour;
                minute = initialMinute;
                second = initialSecond;
        }

        void printTime()
        {
                System.out.println(hour + ":" + minute + ":" + second);
        }

}
```

After finishing the implementation, you ask this question to yourself: " I have done a good job, my implementation looks real good. However, there is one thing that I do not understand. Since, he will be the one using my class, he should be careful, not me! But then, why did he say that my class should be very carefully coded? Is there anything else that I can do?".

Yes! There is actually something that you could do.

Before that, let's see what problems could this version of the Time class cause.
Using the Time class that you have sent, your supervisor executes the following code in his application:

```
class Application
{
        public static void main(String args[])
        {
                Time time1 = new Time();
                time1.hour = 14;
                time1.minute = 65; // accidently!
                time1.second = 22;
        }
}
```

Your supervisor, accidently assigns a value of 65 to the minute field of your Time class. And because of this mistake, the whole application crashes!

Noticing the error, your supervisor comes to you and says: “How did you make a mistake like that !! A minute cannot have a value that is higher than 59! How could you allow it to have such a value!! My application has crashed because of your error!!! You should have protected your class fields!”

Although you think that he is the one that made the mistake, actually what he says is completely correct. As the implementor of the Time class, you are responsible for the fields of your class. Since it doesn’t make sense, nobody must be able to assign a value that is greater than 59 (or negative) to the minute field.

However, until now, you didn’t know how to do that. Right now, it is time to learn how to protect the fields of your class.

It is all about the concept of visibility.
If you were able to say: "I want my minute field to be invisible to any user of this class", then nobody would be able to see that field.

The invisibility could be achieved using the **private** keyword:

```
class Time
{
        private int hour;
        private int minute;
        private int second;

        // constructor...

        // methods...
}
```

Let's see how this change would affect the application class.

Let's try to execute the same code:

```
class Application
{
        public static void main(String args[]
        {
                Time time1 = new Time();
                time1.hour = 14;
                time1.minute = 65;
                time1.second = 22;
        }
}
```

The code snippet above, would not compile due to the compilation errors. Since the fields of the Time class are private, you can no longer access them by simply using the dot notation.

Right now, let's stop and think for a while. We have partially achieved our goal. The supervisor can no longer assign a value of 65 to the minute field, since it is not visible. However, the supervisor right now cannot assign any value to the fields of the Time class. We should, somehow, allow only certain values to be assigned to our fields. But how?

This could be achieved by using methods!
Let's see an example:

```
class Time
{
        ...
        void setMinute(int newMinute)
        {
                // If the value of the parameter is invalid
                if(newMinute > 59 || newMinute < 0)
                {
                        System.out.println("Invalid minute value... It will now get default value 0");
                        minute = 0;
                }
                // If the value of the parameter is OK
                else
                {
                        minute = newMinute;
                }
        }
}
```

We have added a new method to our class file, and it is called setMinute(int).

The setMinute(int) method ensures that the minute field of the Time class, always gets assigned a valid value. If the provided parameter is invalid, then it prints out an error message and assigns a default value of 0.

Right now, we are able to control the values that the minute field can have. However, there is something missing.

How are we going to read the value of the minute field?
We cannot simply say:

System.out.println(time1.minute);

because the minute field is private.

We will again be needing the help of methods.
This time, we will need a method that will return the value of the field.

Here is that method:

```
class Time
{
        ...
        int getMinute()
        {
                return minute;
        }
}
```

Since the minute field is private, we will be using this method to access the minute field of the Time class.

```
class Application
{
  public static void main(String args[]
  {
    Time time1 = new Time();
    time1.setMinute(65);
    System.out.println(time1.getMinute());
    time1.setMinute(32);
    System.out.println(time1.getMinute());
  }
}
```

Output

```
Invalid minute value... It will now get default value 0

0

32
```

Right now, we know that we should use the keyword private, for making our fields invisible to the outside.
What about the opposite? What should we do if we want our fields, constructors or methods to be visible?

We should use the keyword public.
The public elements (fields, constructors or methods) of the class declaration are visible to anyone that uses the class.
For example, we would want the getMinute() method to be visible.
Therefore, the declaration should look like:

```
class Time
{
        ...

        public int getMinute()
        {
                return minute;
        }
}
```

Now, let's trace back the steps that we have followed for protecting the fields of a class:

1) Make sure that all the fields of the class are declared as private.
Example: private int minute

2) Add a new method for each field, that will make sure that only valid values could be assigned to the private field.
Example: the setMinute(int) method

3) For each private field, add another method that will allow the users of this class to read the value of the private field.
Example: the getMinute() method

4) Make sure that your constructors and your newly added methods are declared public, so that everyone may easily see and use them.
Example: public int getMinute() {...}

With these steps in mind, a complete implementation of the Time class would look like:

```
class Time
{
		private int hour;
		private int minue;
		private int second;

		public Time(int initialHour, int initialMinute, int initialSecond)
		{
				setHour(initialHour);
				setMinute(initialMinute);
				setSecond(initialSecond);
		}

		public void setHour(int newHour)
		{
				// If the value of the parameter is invalid
				if(newHour > 23 || newHour < 0)
				{
						System.out.println("Invalid hour value... It will now get default value 0");
						hour = 0;
				}
				// If the value of the parameter is OK
				else
				{
						hour = newHour;
				}
		}

		(continued...)
```

```
public void setMinute(int newMinute)
{
        // If the value of the parameter is invalid
        if(newMinute > 59 || newMinute < 0)
        {
                System.out.println("Invalid minute value... It will now get default value 0");
                minute = 0;
        }
        // If the value of the parameter is OK
        else
        {
                minute = newMinute;
        }
}

public void setSecond(int newSecond)
{
        // If the value of the parameter is invalid
        if(newSecond > 59 || newSecond < 0)
        {
                System.out.println("Invalid second value... It will now get default value 0");
                second = 0;
        }
        // If the value of the parameter is OK
        else
        {
                second = newSecond;
        }
}
```

(continued...)

```
        public int getHour()
        {
                return hour;
        }

        public int getMinute()
        {
                return minute;
        }

        public int getSecond()
        {
                return second;
        }

        public void printTime()
        {
                System.out.println(hour + ":" + minute + ":" + second);
        }

} // end of the class declaration
```

If you had given this class to your supervisor, then your supervisor would have immediately known that he accidently assigned 65 to the minute field. Therefore, he would have the chance to review his application code.

That completes our discussion of the visibility of class elements. Here are some questions that you might be asking:

Q: I can easily see/access the fields of the class, eventhough they are **not** declared as public. Actually, this was the case in the previous examples of this tutorial. How was that possible? If I do not put anything (neither public nor private), does that automatically mean it is public?

A: The answer to this question is a bit more complicated than just a simple yes or no.
You may assume that there are 3 different combinations you can use to declare a field:
1) Public: Everyone can easily see/access the field.
2) Private: Nobody can access the field except the class methods and constructors.
3) Neither public nor private: The fields will be accessible by other classes from the same package. However, from another package it will behave as if it is private. This is called the package access.

Q: Do I have to declare each field as private and associate methods for setting and getting the value of it? Or is it just when I need to control the values of the fields?

A: Java doesn't enforce such a rule. In other words, you may declare your fields to be public.
However, it is a good practice to always have your fields as private and have public set and get methods. The reason:
When you design your class, you may first think that a field may have any value. That means, there is no need to control the value of that field. However, maybe after a certain time such a need may arise. In that case, you would have to change/modify a lot of your code. Therefore, it is always good to have these get/set methods ready, eventhough it may look like:

```
public void setMinute(int newMinute)
{
        minute = newMinute;
}
```

Remember the distinction that we have made among the classes of Java?

- Static classes that you have already been using to write your application code. For example: Reading a value from the user and printing out the value.
- Non-static classes that you have been using throughout the tutorial. You use these classes to define objects such as the Book object. Later, in your static class you may create and use the objects.

Previously, we have indicated that in non-static classes, you will not need the keyword static. Each variable (field) or method are declared without using the keyword static as opposed to the static classes.

What we have said wasn't completely correct. It was there to simplify things before you get the basics of object orientation. You can actually have static elements in your non-static classses.
Here is how:

Let's say you want to keep track of the number of book objects that were created. Here is a code that does that:

```
public class Application
{
      public static void main(String args[])
      {
        int numberOfBooks = 0;

        Book book1 = new Book("The Secret", 2006, " Rhonda Byrne");
        numberOfBooks = numberOfBooks + 1;

        Book book2 = new Book("Stuff on My Cat", 2006, "Mario Garza");
        numberOfBooks = numberOfBooks + 1;

      }
}
```

The code that does the keeping track is in the application. That means, when you give the Book class implementation to your friend, s/he will again have to write the same number tracking code. Wouldn't it be nice if the Book class itself kept track of the number of created objects?

Yes! That can be done using static elements.

A static field in a class declaration means:

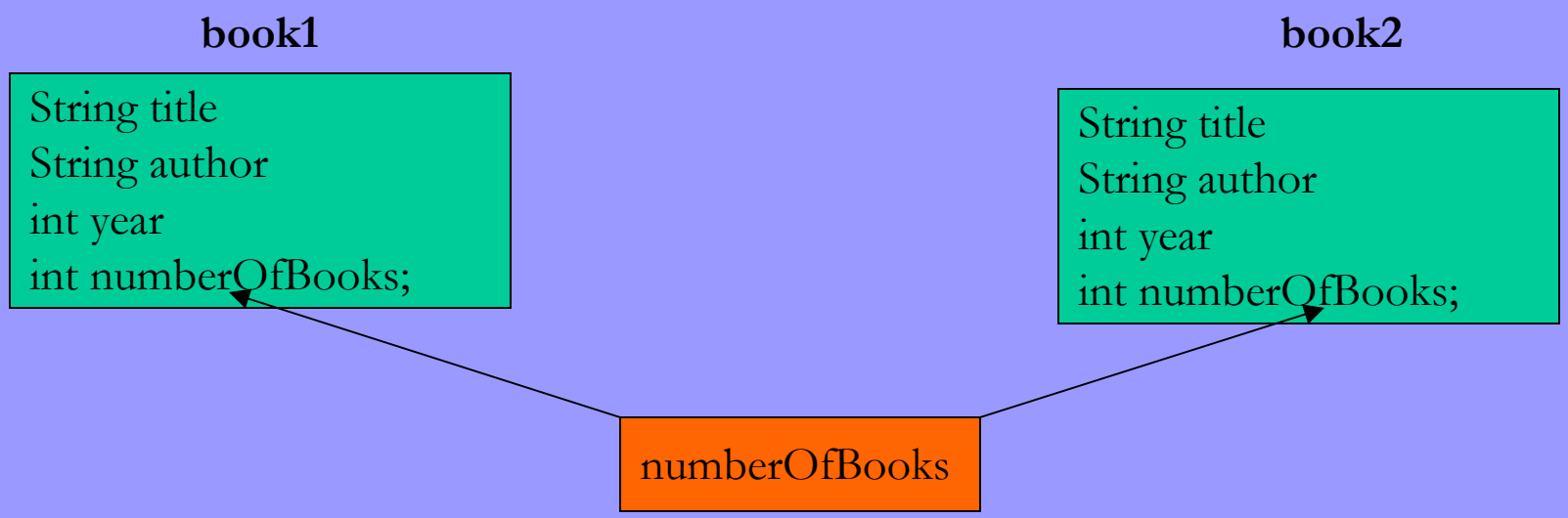
- This field is one and unique. When you create a new object, this field does not get copied! It is a per-class field.
- The newly created objects may access this field. Any update made on this field can be seen by other objects.

An example will make things clear:

```
public class Book
{
        private String title;
        private int year;
        private String author;

        static int numberOfBooks;

        // more code here...

}
```

Let's see what happens as different book objects are created:

```
Book book1 = new Book("The Secret", 2006, " Rhonda Byrne");
Book book2 = new Book("Stuff on My Cat", 2006, "Mario Garza");
```



Although each have separate title, author and year fields, there is only one numberOfBooks field which is shared by the two.

Since there is only one copy of a static field, it is the perfect candidate to store the number of created book objects.
Here is how:

```
class Book
{
        public String title;
        public int year; // using public fields for sake of simplicity
        public String author;

        public static int numberOfBooks;

        Book(String initialTitle, int initialYear, String initialAuthor)
        {
                title = initialTitle;
                year = initialYear;
                author = initialAuthor;

                // everytime a new Book object is constructed
                // increment the counter
                numberOfBooks = numberOfBooks + 1;
        }
}
```

With this code, the keeping track operation is automatically done by the class. So, the application code would look like:

```
public class Application
{
      public static void main(String args[])
      {
        Book book1 = new Book("The Secret", 2006, " Rhonda Byrne");

        Book book2 = new Book("Stuff on My Cat", 2006, "Mario Garza");

        System.out.println("Total number of books: " + Book.numberOfBooks);
      }
}
```

As you may have noticed, for accessing a static field you may use the class name with the dot operator.
That makes a lot of sense because the static field does not belong to any particular instance. Instead, it is a class-wide variable. Therefore, it is logical that you use the class name to access it.

Review of Concepts

- In certain occasions, you want your fields to have a range of values that they can get. Using the visibility settings, it is possible to enforce such a range.
- The values of the private fields cannot be read/changed by simply using the dot notation.
- A good practice is: having your fields private + public set and get methods.

- One could use a static field in a class declaration to make sure that the fiels is a per-class variable. In other words, there is only one copy of that field, which is shared by all the class instances.

Review of Syntax

- For changing the visibility:
  - **private** keyword (e.g. private int year)
  - **public** keyword (e.g. public int year)

- For making sure that only one copy of the field exists:
  - **static** keyword (e.g. static int year)

Exercise

The task is to model a Bike with the following properties:

- A serial ID (represented by an integer value)
- Color – For this exercise, a bike could only get 2 colors. It could either be yellow or red. Other values should be rejected.
- The total number of gears – It could be between 1 and 32.
- The date that it was produced

➢ It must be possible to create a new Bike object given its properties listed above.

➢ Whenever a new bike is created, it should automatically get a serial ID. Furthermore, each assigned serial ID should be unique (like a real life example.) That means, there shouldn't two bikes with the same serial ID. Hint: use of static variables.

➢ The serial number field of a bike cannot be changed! It gets assigned when it is created and does not change afterwards. It can only be read.

Exercise

The production date should be composed of:

- Year – Could only be between 1900 and 2007.
- Month – The valid values are between 1 and 12.
- Day – The valid values are between 1 and 31.

Solution

```
class Date
{
  private int year;
  private int month;
  private int day;

  public Date(int initialYear, int initialMonth, int initialDay)
  {
    setYear(initialYear);
    setMonth(initialMonth);
    setDay(initialDay);
  }

  public int getYear()
  {
    return year;
  }

  public int getMonth()
  {
    return month;
  }

  public int getDay()
  {
    return day;
  }
  (continued...)
```

Solution

```
public void setYear(int newYear)
  {
   if(newYear > 2007 || newYear < 1900)
   {
    System.out.println("Year value is invalid... Setting to default value: 0");
    newYear = 0;
   }
   else
   {
    year = newYear;
   }
  }
  public void setMonth(int newMonth)
  {
   if(newMonth > 12 || newMonth < 1)
   {
    System.out.println("Month value is invalid... Setting to default value: 0");
    newMonth = 0;
   }
   else
   {
    month = newMonth;
   }
  }
  (continued...)
```

Solution

```
public void setDay(int newDay)
 {
  if(newDay > 31 || newDay < 1)
  {
   System.out.println("Day value is invalid... Setting to default value: 0");
   newDay = 0;
  }
  else
  {
   day = newDay;
  }
 }
} // end of class declaration
```

Solution

```
public class Bike
{
  private int serialID;
  private String color;
  private int numberOfGears;
  private Date productionDate;

  private static int IDGenerator = 0;

  public Bike(String initialColor, int totalGears, Date date)
  {
    setColor(initialColor);
    setNumberOfGears(totalGears);
    setProductionDate(date);

    serialID = IDGenerator;
    IDGenerator = IDGenerator + 1;
  }
  (continued...)
```

Solution

```
public String getColor()
 {
  return color;
 }

 public int getNumberOfGears()
 {
  return numberOfGears;
 }

 public Date getProductionDate()
 {
  return productionDate;
 }

 public int getSerialID()
 {
  return serialID;
 }
(continued...)
```

Solution

```
public void setColor(String newColor)
 {
  if(! (newColor.equals("Red") || newColor.equals("Yellow")) )
  {
    System.out.println("Color value is invalid... Setting to default value: No Color");
    color = "No Color";
  }
  else
  {
    color = newColor;
  }
 }
 public void setNumberOfGears(int gears)
 {
  if(gears > 32 || gears < 1 )
  {
    System.out.println("Gear value is invalid... Setting to default value: 0");
    numberOfGears = 0;
  }
  else
  {
    numberOfGears = gears;
  }
 }
 (continued...)
```

Solution

```
public void setNumberOfGears(int gears)
 {
  if(gears > 32 || gears < 1 )
  {
    System.out.println("Gear value is invalid... Setting to default value: 0");
    numberOfGears = 0;
  }
  else
  {
    numberOfGears = gears;
  }
 }
} // end of class declaration
```