KIT101 Notes — 11 — Creating Your Own Data Types (Classes)

James Montgomery

Revised: 2018-02-16

References: L&L 4.1–4.5, 7.1

There are varying degrees of object-oriented (OO) design. This section (i.e., web page) of the *Introductory Programming Notes* deals with defining new data types (that is, classes), while subsequent sections deal with issues of making the objects you define even more self-sufficient (i.e., with more code to manage the data they hold) and how you can model and implement larger programs.

Basic OO: Classes as Data

The simplest form of object-orientation is defining a new data type that comprises several primitive (or other object) types of data. For instance, a 'Person' type might have a String-valued name, int-valued age and int-valued height, among other properties. In order to manage that data you'll also define a number of methods (think of the Turtle and its `getColor()` and `setColor()` methods, which report or change the internal state of the Turtle object). A very high-level template for an object is this:

```
public class ClassName {
    variable (property) declarations
    method declarations
}
```

where the *variable declarations* are the individual items of data (integers, real-numbers, booleans, Strings or other kinds of objects) and the *method declarations* are all the methods for interacting with the object.

We can expand this template a little to give a suggested structure (the links jump further down the document):

```
public class ClassName {
    variable (property) declarations

    constructor declaration(s)

    getter and setter methods

    public String toString() {
        code to generate a String representation of this object
    }

    other methods
}
```

The following sections explore each of these in turn with the working example of a class to represent a **rectangle**.

Tip: A common development pattern for implementing a new class

The following approach is general purpose and suitable for use even beyond your time studying for your degree. In fact, only the mechanism for implementing the "driver" class defined below will change (if you're feeling adventurous, spend a few minutes reading about the purpose of products like JUnit).

1. Plan the new class
    - Data storage
    - Methods (what they'll do [achieve], their return type and any parameters they'll have)
2. Create a "driver" program that contains only a `main()` method. If the new class is called `Mini` then you could call your driver program `MiniDriver` (for example).
3. Begin implementing the new class in its own .java file (use DrJava to help get you started or use the template above).
4. Enter the following loop, while you are not finished:
    - Write a method in the class source file
    - Call the method from the driver program
    - Observe the results (are they what you expect?)

- Fix any errors

Here is a (nearly) minimal example of a driver program and class that can be tested

Driver.java

```
public class Driver {
    public static void
main(String[] args) {
        int value;
        Simple s = new
Simple();
        value = s.getValue();

System.out.println("value is "
+ value);
    }
}
```

Simple.java

```
public class Simple {
    private int lonelyVar;

    public int getValue() {
        return lonelyVar;
    }
}
```

Variable declarations

The example Rectangle class will be defined by two integer properties, its width and height (it does have a location in space, for simplicity. So we can start to define the new class:

```
/**
 * A rectangle with a width and height.
 * @author Anonymous
 */
public class Rectangle {
    private int width; //width of rectangle
    private int height; //height of rectangle

}
```

The keyword **`private`** is optional but highly recommended. It means that those variables are accessible inside the code for Rectangle but nowhere else.

> **Tip:** The variables defined within a class are also referred to as *data fields*.

The above code would be saved to its own file, Rectangle.java. As it does not have a main() method this is not a program in its own right and cannot be run.

Methods in class declarations

The template for a method that is part of a new class of objects is this, which differs from the methods you write in the main program by omitting `static`:

```
/**
 * comments describing the method.
 */
public return_type identifer ( parameter_list ) {
    local variable declarations
    code to do the work
    return statement (if return_type is not void)
}
```

Note: When (and when not) to use `static`:

1. If the method you are defining is in the same source file as `main()` and you want to be able to call it directly from `main()` then include the word `static` as in `public static void myMethod()`
2. If you are defining a new object type, then do not include the word `static`.

Tip: Stub Methods

In addition to the general pattern for implementing a new class described above, it can also be useful to implement "stub methods". These consist of a method header (specifying the access level, return type, name and parameters) and might print out a tracing or debug message (and may return a dummy result if their return type is not `void`). They should also be preceded by a comment saying what the method is to do. Think of stub methods like installing buttons on the front panel of some complex machine before doing the wiring behind.

Constructors

Given the initial definition of Rectangle we could create a program with a main() method that creates an object of type Rectangle:

```
public class ActualProgram {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle();
    }
}
```

With no other code to perform initialisation, object properties are initialised to default values

based on their type:

Data type	Default property value
`int`, `double`	0
`char`	the null character (0)
`boolean`	**`false`**
any class type, including String	**`null`**

So the above main() method would create a Rectangle object with width and height both 0, which is not very useful. The role of a *constructor* is to provide necessary initialisation.

A constructor is a special method that is used to set up a newly created object and ensure it is ready to be used in a program. Its main role is to set the values of the object's data fields to sensible initial values.

Although a constructor is a kind of method, there are particular restrictions on how it is defined. Specifically, a constructor:

- has the same name as the class
- does not return a value
- has no return type (not even `void`)
- can have parameters

```
public ClassName(parameter list) {
    statement(s)
}
```

We could define a constructor for Rectangle that accepts values for its width and height. Somewhere after the variable declarations:

```
/** Creates a new Rectangle with the given width and height. */
public Rectangle(int w, int h) {
    width = w;
    height = h;
}
```

and then in ActualProgram's main() method we would need to pass some values:

```
Rectangle rect = new Rectangle(100, 200);
```

Tip: While you *can* define the parameters of a constructor to have the same names as the object's properties, they will then mask (that is, hide) the properties within the constructor. But you can refer to the object's variables by prefixing their name with **`this.`**, as in **`this.`**`width`. The keyword **`this`**, which we'll look at more closely later, refers to 'the object that you are currently inside'.

Tip: You've already encountered constructors that require values in order to work: the Scanner's constructor requires that you give it a source of text to read from. Passing `System.in` provides this necessary data.

Getters and Setters

Because the data fields (properties) are marked **`private`** there is no way to retrieve them from a Rectangle object yet (writing `rect.width` inside main() would be a compiler error). So each property should have a method to *get* its value and may have a method to *set* its value. These methods are consequently known as 'getters' and 'setters'.

The general form of a getter is:

```
public type getProperty() {
    return property;
}
```

and the general form of a setter is:

```
public void setProperty(type value) {
    property = value;
}
```

So we could define getters and setters for width and height in the Rectangle class as:

```
public class Rectangle {
    // Variable declarations and constructor code were here

    /** Returns the Rectangle's width. */
    public int getWidth() {
        return width;
    }

    /** Sets the Rectangle's width. */
    public void setWidth(int w) {
        width = w;
```

```
    }

    /** Returns the Rectangle's height. */
    public int getHeight() {
        return height;
    }

    /** Sets the Rectangle's height. */
    public void setHeight(int h) {
        height = h;
    }
}
```

Tip: A *read only* property only has an associated getter, but no setter. It may be modified by other code inside the class definition but not directly modified from outside.

That may look like a lot of unnecessary code, and it can be tedious to create all the necessary getters and setters for a new class of object. However, the code inside doesn't *have* to be as simple as what is shown in this example. We could, for example wrap the code inside `setWidth()` with `if (w > 0) { }` so that any attempt to set the width to a negative value or zero would be ignored. This allows us to control what values the internal data can take.

toString

Every object type in Java has a method called `toString`, which is responsible for generating a String representation of that object. It is called automatically when the system needs a String represenation of an object, such as when passing the object to `System.out.println()`. Its header is:

```
public String toString()
```

When you define a new class it will inherit a default implementation (the code for which is burried in the Java Standard Library), which produces an uninteresting and non-human friendly representation consisting of the class's name and a sequence of hexadecimal digits. So currently this program would not produce nice output:

```
public class ActualProgram {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(100, 200);

        System.out.println("The rectangle is " + rect);
    }
}
```

But you can provide your own implementation of `toString()` which returns a meaningful String

(where meaningful depends on your needs). For the Rectangle we could define:

```
public class Rectangle {
    //other code was here

    /** Returns a String representation of this Rectangle: a width x
height Rectangle. */
    public String toString() {
        return "a " + width + " x " + height + " Rectangle";
    }
}
```

Running `ActualProgram` again would now produce the following:

```
The rectangle is a 100 x 200 Rectangle
```

Other methods

Other methods are any other abilities we wish to give the object. For example, we may want a Rectangle to be able to calculate its area, so we can define a method, inside **`public class`** `Rectangle { }`:

```
/** Returns the area of the Rectangle. */
public int area() {
    return width * height;
}
```

Note: This function doesn't take any parameters because it uses the values of the instance variables *width* and *height*.

An expanded template

Combining all of the above we can define an expanded template, showing you the parts you can or should change and those that you can treat as fixed:

```
/**
 * Briefly describe the class.
 * @author Put the author's name here
 */
public class ClassName {
    private type property1;
```

```
    /** Creates a new ClassName... */
    public ClassName( parameters ) {
        statements to set up object
    }

    /** Returns the ClassName's property1. */
    public type getProperty1() {
        return property1;
    }

    /** Sets the ClassName's property1. */
    public void setProperty1( type p1 ) {
        property1 = p1;
    }

    other methods as required

    /** Returns a String representation of this ClassName. */
    public String toString() {
        code to generate a String representation of this object
    }
}
```

So the full Rectangle class implementation would be:

```
/**
 * A rectangle with a width and height.
 * @author Anonymous
 */
public class Rectangle {
    private int width; //width of rectangle
    private int height; //height of rectangle

    /** Creates a new Rectangle with the given width and height. */
    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    /** Returns the Rectangle's width. */
    public int getWidth() {
        return width;
    }

    /** Sets the Rectangle's width. */
```

```
    public void setWidth(int w) {
        width = w;
    }

    /** Returns the Rectangle's height. */
    public int getHeight() {
        return height;
    }

    /** Sets the Rectangle's height. */
    public void setHeight(int h) {
        height = h;
    }

    /** Returns the area of the Rectangle. */
    public int area() {
        return width * height;
    }

    /** Returns a String representation of this Rectangle: a width x
height Rectangle. */
    public String toString() {
        return "a " + width + " x " + height + " Rectangle";
    }

}
```

Documenting a class: class diagrams

One way of succinctly summarising a new class it to use a class diagram, where the new type of data is represented by a box with sections for its name, attributes (properties) and methods. The Rectangle class could be depicted as:

Rectangle
`int width` `int height`

`int getWidth()`
`void setWidth(int w)`
`int getHeight()`
`void setHeight(int h)`
`int area()`
`String toString()`

This could then be supplemented by descriptions or algorithms for each of the key methods, similar to how we have documented methods in the tasks you have seen so far.

Worked example: A bank account

Let's work through a complete example, based on a tutorial task from a previous year (the equivalent portfolio task this year is a little simpler). Imagine you would like to represent a *bank account* as part of a system you are designing. Before you look further down, work through this activity:

Activity: Modelling a Bank Account

What kind of *information* is needed to describe a bank account? What *type* does each piece of information have? Write down a list.

What kind of *operations* would you (if you were a bank) want to perform on that data? Think about the core business of keeping track of money but also other things, like customers changing their names. Write down a list of these operations.

The remainder of this section works through one *particular* design, but you can always elaborate on this with features you thought of above.

The following class diagram represents *one* set of data and methods that a bank account class could support. It's actually very limited, with no support for the type of account or its interest rate (needed by both savings and credit accounts). But it will suffice for the demonstration.

BankAccount

`String name`
`int number`
`double balance`

`String getName()`
`void setName(String accountName)`
`int getNumber()`
`double getBalance()`
`String toString()`
`void adjustBalance(double byAmount)`
`void deposit(double amount)`
`void withdraw(double amount)`

In this plan we have made *name* a modifiable property (as clients can request this be changed and *number* a read only property (as this is fixed at account creation). Also, rather than allowing arbitrary changes to *balance* we have methods suited to a bank account to adjust the balance (add a positive or negative amount) and also for the common user actions of depositing or withdrawing money.

Here is a detailed description of the methods to be implemented and their expected behaviour:

Method	Returns	Parameters	Function

BankAccount	*n/a*	String name, int number, double initialBal	The constructor, which has parameters for the account name, number and initial balance, and assigns those values to the instance variables.
getName	String	*none*	*Returns* the **name** of the BankAccount
setName	void	String accountName	*Sets* the **name** of the BankAccount to **accountName**
getNumber	int	void	*Returns* the BankAccount's **number**
getBalance	double	*none*	*Returns* the current **balance**
toString	String	*none*	*Returns* a new String that includes the BankAccount's name, number and balance
adjustBalance	void	double byAmount	*Updates* the **balance** by adding **byAmount** to it (byAmount may be negative)
deposit	void	double amount	*Updates* the balance by calling **adjustBalance** if amount is not negative; displays an error otherwise
withdraw	void	double amount	*Updates* the balance by calling **adjustBalance** if amount is positive and no greater than balance; displays errors for negative values and values that are too large

Create a Bank (demo)

Activity: Create a small program to test drive the bank account while it is developed.

In DrJava create a new program using the following code. You will modify this file to test each BankAccount feature as you implement it.

```
/**
 * BankAccount tester.
 */
public class BankDemo {

    public static void main(String[] args) {
        BankAccount account;

        //TODO Instantiate account using new with BankAccount's
constructor
```

```
        //TODO Call its methods and display results to test if it
works as expected
    }

}
```

Implement and test

Activity: Add data

In a new source file start defining the BankAccount class and add instance variable declarations to it for the account name, number and balance.

```
public class BankAccount {
    private String name;    //account name; usually the customer's
name
    private int number;     //account number, given by the bank
    private double balance; //current balance in dollars and cents

    //Constructor and methods will go here
}
```

We can't test much yet, so let's implement each of BankAccount's methods in turn. After writing each one, we can add some code to BankDemo to check that it worked (we can't check much after implementing the constructor other that the program runs, but for each of the others we can either change the BankAccount object or check what data it holds).

Activity: Implement the constructor

The constructor should have three parameters: `String name, int number, double initialBal`. Assign their values to the instance variables. Use `this.name` to refer to the instance variable `name` instead of the *parameter* called `name`.

This is a fairly typical constructor. Note that it *could* do some checking to make sure the initial balance is zero or higher.

```
public BankAccount(String name, int number, double initialBal) {
    this.name = name;
    this.number = number;
    this.balance = initialBal;
}
```

Back in BankDemo we could add the following:

```
account = new BankAccount("Henry Jones", 12345678, 1000);
```

But how can we modify those values after the BankAccount has been created?

Activity: Getting and setting properties

Implement the getters (and one setter) for account name, account number and balance. The general forms of getters and setters are given above.

Remember, the properties of BankAccount are *name*, *number* and *balance*.

```
public String getName() {
    return name;
}

public void setName(String accountName) {
    name = accountName;
}

public int getNumber() {
    return number;
}

public double getBalance() {
    return balance;
}
```

You could now add some more code to BankDemo to test this functionality out. Here's a subset of things you could try:

```
System.out.println("Account name is " + account.getName());
account.setName( account.getName + " Jnr");
System.out.println("Account name is now " + account.getName());
System.out.println("Account balance is $" + account.getBalance());
```

Activity: Implement toString()

Add the following to the BankDemo program at any point after you've created the BankAccount object:

```
System.out.println("The BankAccount object is: " + account);
```

What happens when you run the program? What do you think is happening? Check above if

you need a reminder.

Add a new method based on the following template:

```
public String toString() {
    return "If you can read this when running the program you've not
finished the activity";
}
```

and replace that String with any combination of text and the instance variables you like.

This is a basic example that includes the number, name and balance, with explanatory text, spread across several lines (The "{.java} 'escape sequence' represents the newline character):

```
public String toString() {
    return "Account number: " + number + "\nAccount name: " + name +
"\nBalance: $" + balance;
}
```

More challenging activities

Activity: Completing the implementation

A static bank account isn't very useful. It's time to implement the remaining three methods `adjustBalance`, `deposit` and `withdraw`. As above, let's implement them one at a time and add suitable calls in BankDemo to see if they work as expected.

A basic implementation of `adjustBalance`:

```
public void adjustBalance(double byAmount) {
    balance += byAmount;
}
```

To test this, display the balance before and after calling this method.

```
public void deposit(double amount) {
    if (amount > 0) {
        adjustBalance(amount);
    } else {
        System.out.println("Error: Cannot deposit negative amount");
    }
}
```

Note that amount is made negative *inside* this method because the customer is withdrawing a positive dollar value.

```
  public void withdraw(double amount) {
      if (amount <= 0) {
          System.out.println("Error: Cannot withdraw a negative
  amount");
      } else if (amount <= balance) {
          adjustBalance(-amount);
      } else {
          System.out.println("Error: Withdrawal amount $" + amount + "
  greater than balance of $" + balance);
      }
  }
```

Here are complete versions of BankAccount...

```
/**
 * A simple bank account, with an account name, account number and
balance.
 * Account number is read only, while the balance is modified only
indirectly by
 * adding or substracting amounts.
 *
 * This implementation includes many debugging statements to report on
what is
 * happening when certain methods are called.
 *
 * @author James Montgomery
 * @version April 2016
 */
public class BankAccount {
    private String name;    //account name; usually the customer's name
    private int number;     //account number, given by the bank
    private double balance; //current balance in dollars and cents

    /** Creates a new BankAccount. */
    public BankAccount(String name, int number, double initialBal) {
        this.name = name;
        this.number = number;
        this.balance = initialBal;
    }

    /** Returns the account name. */
    public String getName() {
        return name;
    }

    /** Sets the account name to the new value. */
    public void setName(String accountName) {
        name = accountName;
    }
```

```
    /** Returns the account number. */
    public int getNumber() {
        return number;
    }

    /** Returns the current balance. */
    public double getBalance() {
        return balance;
    }

    /** Returns a String containing the account number, name and current
balance. */
    public String toString() {
        return "Account number: " + number + "\nAccount name: " + name +
"\nBalance: $" + balance;
    }

    /**
     * Adjusts the current balance by the given amount.
     * Does NOT check if adjustment keeps balance >= 0.
     * @param byAmount - the amount to add to the current balance.
     */
    public void adjustBalance(double byAmount) {
        System.out.println("Adjusting balance of account " + number + "
by adding " + byAmount + " to " + balance);
        balance += byAmount;
    }

    /**
     * Adds the given amount to the account balance, provided it is
positive.
     * @param amount - the positive amount to deposit into the account
     */
    public void deposit(double amount) {
        if (amount > 0) {
            System.out.println("Depositing $" + amount + " into account
" + number);
            adjustBalance(amount);
        } else {
            System.err.println("Error: Cannot deposit negative amount");
        }
    }

    /**
     * Subtracts the given amount from the account balance provided
that:
     * (1) it is a positive amount and (2) it is no larger than the
current
     * balance.
     * @param amount - the positive amount to withdraw from the account
     */
    public void withdraw(double amount) {
        if (amount <= 0) {
```

```
            System.err.println("Error: Cannot withdraw a negative
amount");
        } else if (amount <= balance) {
            System.out.println("Withdrawing $" + amount + " from account
" + number);
            adjustBalance(-amount);
        } else {
            System.err.println("Error: Withdrawal amount of $" + amount
+ " greater than balance of account " + number + " ($" + balance + ")");
        }
    }

}
```

...and BankDemo.

```
/**
 * Sample solution illustrating an application to test the newly defined
 * BankAccount class.
 */
public class BankDemo {

    public static void main(String[] args) {
        BankAccount account;

        //Test BankAccount basics of holding data and modifying the
account name
        account = new BankAccount("Scrooge McDuc", 1024, 1000.50);
        System.out.println("New account's name is " +
account.getName());
        System.out.println("New account's number is " +
account.getNumber());
        System.out.println("New account's balance is " +
account.getBalance());

        System.out.println("Changing account's name to " +
account.getName() + 'k');
        account.setName(account.getName() + 'k');
        System.out.println("Account's name is now " +
account.getName());

        //Remember, toString() is called automatically whenever a String
        // representation of an object is required
        System.out.println("Testing BankAcount.toString() result:\n" +
account);

        //Test various balance adjusting methods and their robustness to
invalid values
        System.out.println();
        System.out.println("Testing adjustBalance, deposit and withdraw
with a variety of values");
```

```
        account.adjustBalance(1000);
        System.out.println("Balance is now $" + account.getBalance());
        System.out.println();

        account.deposit(10);
        System.out.println("Balance is now $" + account.getBalance());
        account.deposit(-10); //should report an error
        System.out.println("Balance is now $" + account.getBalance());
        System.out.println();

        account.withdraw(50);
        System.out.println("Balance is now $" + account.getBalance());
        account.withdraw(-50); //should report an error
        System.out.println("Balance is now $" + account.getBalance());
        account.withdraw(5000); //should report an error
        System.out.println("Balance is now $" + account.getBalance());
    }

}
```
