**Carleton University**
**Department of Systems and Computer Engineering**
**SYSC 2004 - Object-Oriented Software Development - Winter 2015**

# Lab 7 - Inheritance: Template Methods and Hook Methods, Abstract Classes and Abstract Methods

## Objective

To develop an inheritance hierarchy of classes that model different types of chess pieces.

## Attendance/Demo

To receive credit for this lab, you must demonstrate your work. **Also, you must submit your lab work to cuLearn**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will grade the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before Lab 8.**

## References

*Objects First with Java*, Chapters 8, 9 and 10.

## Getting Started

**Step 1:** Create a new folder named Lab 7.

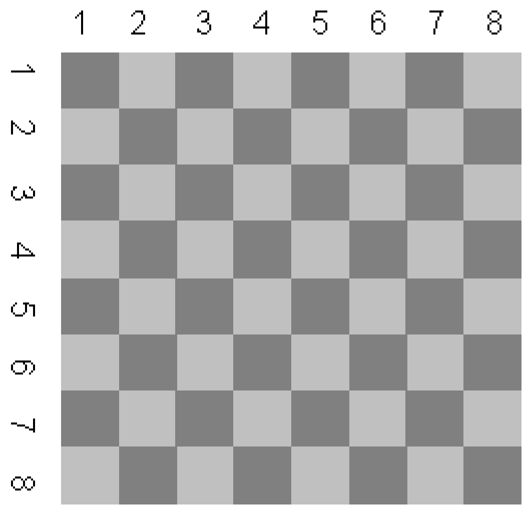
**Step 2:** Download chess.zip to your Lab 7 folder.

**Step 3:** Extract the *chess* project from the zip file into a project folder named chess.

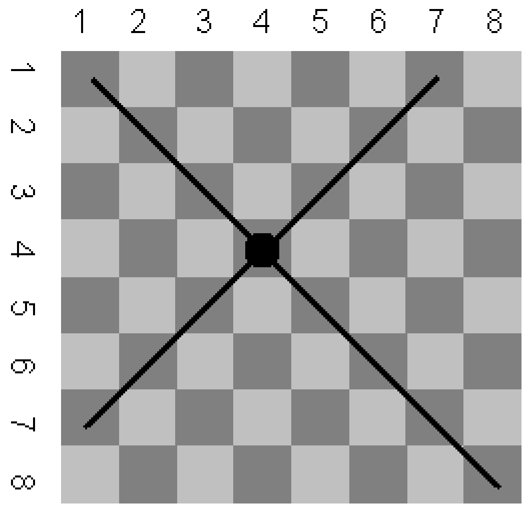
**Step 4:** Open the chess folder. Double-click on the BlueJ project file, which is named package. This will launch BlueJ and open the *chess* project.

## Background - Modelling Chess Pieces

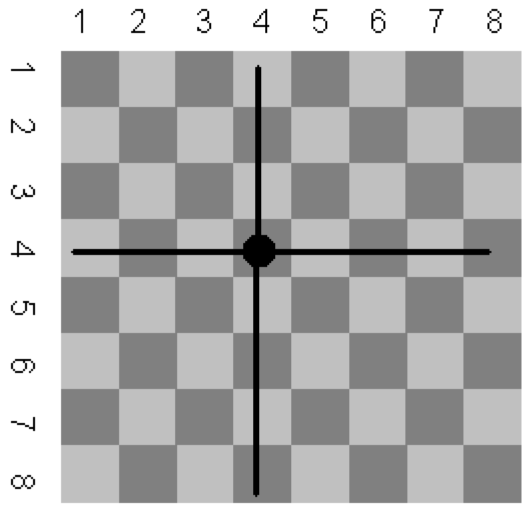
A chessboard is made up of 64 squares arranged in 8 rows and 8 columns. For the purposes of this lab, the rows are numbered 1 to 8, top to bottom, and columns are numbered 1 to 8, left to right, as shown in the following diagram.



Two of the pieces in chess are the bishop and the rook. A bishop moves diagonally, so the bishop located at square (4, 4), as shown below, can move into any square covered by the thick black lines.



A rook moves horizontally and vertically, so the rook located at square (4, 4), as shown below, can move into any square covered by the thick black lines.



The *chess* project contains incomplete implementations of three classes: `ChessPiece`, `Bishop` and `Rook`. `ChessPiece` is called an *abstract class*, because we never intend to create instances of this class. Instead, it's sole purpose it to be a superclass for *concrete classes* that model specific chess pieces.

Bishops and rooks are modelled by classes `Bishop` and `Rook`, respectively. These classes are subclasses of `ChessPiece`. In addition to a constructor, both of these classes will define a method called `canMoveTo`, which determines whether the piece can move from its current location to a specified square. As described earlier, a bishop and a rook don't move the same way, so we can't have a single definition of this method in `ChessPiece` that is inherited by both `Bishop` and `Rook` (and any other subclasses of `ChessPiece` that we develop in the future). Instead, a specialized version of `canMoveTo` must be defined in each subclass of `ChessPiece`.

**Implementing Classes that Model Chess Pieces**

Read all of the following instructions carefully before you write any code.

Your task is to complete the implementation of `ChessPiece`, `Bishop` and `Rook`, subject to the following constraints:

- `ChessPiece` declares four fields (instance variables): `name`, `isWhite`, `column` and `row`. These fields have `private` visibility. Don't change the visibility of these fields. If methods in `Rook` and `Bishop` need to obtain the values stored in these fields, they must call accessor methods that are defined in `ChessPiece`.

- You are NOT permitted to define additional fields in `ChessPiece`, `Bishop` and `Rook`. `Bishop` and `Rook` must inherit all their fields from `ChessPiece`.

- You are NOT permitted to change the signatures of any of the methods defined in the classes; i.e., add or remove parameters or change the types of any parameters.

- You are NOT permitted to define methods other than the ones specified in the source code provided to you.
- Use the constants `MIN_ROW`, `MAX_ROW`, `MIN_COL` and `MAX_COL` in code that checks the minimum and maximum row and column numbers. These constants are defined as class variables in `ChessPiece` and are inherited by the subclasses. **Do not hardwire the values 1 and 8 into your code.**
- When implementing the classes, ignore the possibility that other pieces might be on the board.

**Exercise 1 - Implementing class `ChessPiece`, Part 1**

**Step 1:** Open class `ChessPiece` in the editor.

**Step 2:** The constructor currently contains one statement:

```
this.name = this.getClass().getName();
```

This statement initializes field `name` with the name of the object's class. (This field is used by the `toString` method, which has been defined for you.) Read the constructor's javadoc comment, and finish the implementation of this method.

**Step 3:** Read the javadoc comments for methods `isWhite`, `getColumn` and `getRow`, and finish the implementation of these methods. Do not modify `toString`.

**Step 4:** Create a few different `ChessPiece` objects and place them on the object bench. Some of these pieces should be white and others should be black. Each piece should be located in a different square from the others. Use BlueJ's object inspector to verify that the fields of each `ChessPiece` object have been initialized correctly. Interactively call the `isWhite`, `getColumn` and `getRow` methods and verify that the values they return are correct.

**Exercise 2 - Implementing class `Rook`**

**Step 1:** Open class `Rook` in the editor.

**Step 2:** Read the constructor's javadoc comment and implement the body of the constructor.

**Step 3:** Run the unit tests in class `RookTest`. Tests `testCanMoveTo` and `testMoveTo` will fail, because you haven't implemented the methods that these test cases test. The `testToString` method should pass.

**Step 4:** Read the javadoc comment for `canMoveTo` in `Rook` and write the method. Run the unit tests in `RookTest`. The `testCanMoveTo` method should now pass. If this test fails, make the necessary changes to `Rook` and rerun the unit tests. The `testMoveTo` method will fail - you'll take care of this in the next exercise.

**Exercise 3 - Implementing class `ChessPiece`, Part 2**

**Step 1:** The `moveTo` method in class `ChessPiece` will contain the algorithm that moves a chess piece to a specified location on the chessboard, if it can legally do so. To determine if the requested move is legal, `moveTo` must call `canMoveTo`. The `canMoveTo` method determines whether the piece that a subclass models can be moved to the desired new location. As explained earlier, because different types of chess pieces move in different ways, each subclass of `ChessPiece` must contain a definition of `canMoveTo` that is appropriate for the subclass.

This is an example of a common code pattern in object-oriented programming, in which a method defined in a superclass (known as the *template method*) requires the help of a method defined in a subclass (known as a *hook method*). In this project, the `moveTo` method in `ChessPiece` is the template method, and the `canMoveTo` method that it calls is the *hook method*.

In Exercise 2, you implemented `canMoveTo` in `Rook`, so after you've implemented `moveTo` in `ChessPiece`, you will be able to move `Rook` objects.

Read the javadoc comment for `moveTo` in `ChessPiece`, and write the method.

**Important:**

- The `moveTo` method is NOT permitted to use Java's `instanceof` operator (which has not been covered in class) to determine whether the method is attempting to move a `Bishop` object or a `Rook` object.
- The `moveTo` method is NOT permitted to use access the object's `name` field (or call `getClass().getName()`) to determine whether the method is attempting to move a `Bishop` object or a `Rook` object.
- The `moveTo` method must call `canMoveTo` to determine if the requested move is valid.
- Don't modify the `canMoveTo` method in class `ChessPiece`. We'll work with this method in Exercise 6.

**Step 2:** Run the unit tests in `RookTest`. If you've implemented `moveTo` in `ChessPiece` correctly, the `testMoveTo` method in `RookTest` should pass. If this test fails, make the necessary changes to `ChessPiece` or `Rook` and rerun the unit tests.

**Exercise 4 - Implementing class `Bishop`**

**Step 1:** Open class `Bishop` in the editor.

**Step 2:** Read the constructor's javadoc comment and implement the body of the constructor.

**Step 3:** Run the unit tests in class `BishopTest`. Tests `testCanMoveTo` and `testMoveTo` will fail (because you haven't implemented `canMoveTo` for class `Bishop`). The `testToString` method should pass.

**Step 4:** Read the javadoc comment for `canMoveTo` in `Bishop` and write the method. Run the unit tests in `BishopTest`.

- The `testCanMoveTo` method in `BishopTest` should now pass. If the test fails, make the necessary changes to `Bishop` and rerun the unit tests.

- If you've implemented `moveTo` in `ChessPiece` correctly, the `testMoveTo` method in `BishopTest` should pass. If this test fails, make the necessary changes to `ChessPiece` or `Bishop` and rerun the unit tests.

**Exercise 5 - Review: Dynamic Method Lookup**

Create instances of `ChessPiece`, `Rook` and `Bishop` and place the objects on the object bench. Recall that the `toString` method is defined in `ChessPiece`, but is not defined in `Rook` or `Bishop`.

- Read the definition of `toString` in `ChessPiece`, and predict the result of calling `toString` on a `ChessPiece` object; in other words, what `String` do you think this method will return?

- Predict the result of calling `toString` on a `Rook` object. Will there be a run-time error (because `toString` isn't defined in `Rook`)? If you think `toString` will run, what `String` do you think this method will return?

- Predict the result of calling `toString` on a `Bishop` object. Will there be a run-time error? If you think `toString` will run, what `String` do you think this method will return?

- Create instances of `ChessPiece`, `Rook` and `Bishop` and place the objects on the object bench. Interactively call `toString` on the three objects. Were your predictions correct?

**Exercise 6 - Abstract Methods and Abstract Classes**

**Step 1:** Scroll through `ChessPiece` until you find the definition of the `canMoveTo` method, which looks like this:

```
public boolean canMoveTo(int desiredRow, int desiredColumn)
{
    return false;
}
```

A method that always returns `false` is smelly code. Now that you've finished Exercises 2 and 4, your `Rook` and `Bishop` classes override this method; that is, each subclass defines an appropriate version of `canMoveTo`, so you might wonder why we bothered to define `canMoveTo` in class `ChessPiece`.

Comment out the method by enclosing it in `/* */`. It should now look like this:

```
/*
```

```
public boolean canMoveTo(int desiredRow, int desiredColumn)
{
     return false;
}
*/
```

**Step 2:** Compile `ChessPiece`. When this class is compiled, a compilation error will occur:

```
cannot find symbol - method canMoveTo(int, int)
```

Because `moveTo` (in `ChessPiece`) contains a call to `canMoveTo`, Java requires that `canMoveTo` be defined in `ChessPiece` or its superclass, `Object`. This definition is needed by the Java compiler so that it can verify that the argument list in the method call is correct; that is, contains two arguments of type `int`. Because `canMoveTo` in `ChessPiece` has been commented out, the Java compiler cannot determine if the method call is correct, so it issues the compilation error.

In other words, `canMoveTo` is defined in `ChessPiece` solely to prevent a compilation error. This method will never be executed at run-time. When `moveTo` is called on a `Rook` object, `moveTo` calls the `canMoveTo` method in `Rook`. When `moveTo` is called on a `Bishop` object, `moveTo` calls the `canMoveTo` method in `Bishop`. The `canMoveTo` method in `ChessPiece` will never be called by `moveTo`, but as we've observed, we can't simply comment out this method in `ChessPiece`.

**Step 3:** We'll now remove the code smell. Delete the definition of `canMoveTo` in `ChessPiece` and replace the deleted code with the following declaration:

```
public abstract boolean canMoveTo(int desiredRow,
                                  int desiredColumn);
```

The semicolon at the end of the method declaration is mandatory!

This is how we declare an *abstract method* in Java. An abstract method has no body (no `{}'`s after the method header). The Java compiler will use this declaration when it compiles the `moveTo` method in `ChessPiece` to verify that the call to `canMoveTo` is correct.

Defining an abstract method in `ChessPiece` does one additional thing: it specifies that concrete subclasses of `ChessPiece` must provide a complete definition of this method. If we forgot to define `canMoveTo` in `Rook` or `Bishop` (or any other subclass of `ChessPiece`), a compilation error would occur when we compiled the subclass.

**Step 4:** Compile `ChessPIece`. Another compilation error will occur:

```
ChessPiece is not abstract and does not override abstract method
canMoveTo(int, int) in ChessPiece.
```

When a class contains one or more abstract methods, we must also declare the class to be abstract. Edit the class header to look like this:

```
public abstract class ChessPiece
```

Compile the project. There should be no compilation errors.

**Step 5:** Abstract classes cannot be instantiated. In the case of `ChessPiece`, this is exactly what we want (what chess piece would a `ChessPiece` object represent)? Try to instantiate a `ChessPiece` object by right-clicking on the `ChessPiece` class icon. Does the pop-up menu contain a command to allow you to create a new `ChessPiece` object?

**Wrap-Up**

1. With one of the TAs watching, run the JUnit tests in `RookTest` and `BishopTest`. The TA will note how many test cases pass. The TA will review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.

2. The next thing you'll do is package the project in a *jar* (Java archive) file named chess.jar. To do this:

   2.1. From the menu bar, select Project > Create Jar File... A dialog box will appear. Click the Include source and Include BlueJ project files check boxes. A check-mark should appear in each box. Do not modify the Main class field.

   2.2. Click Continue. A dialog box will appear, asking you to specify the name for the jar file. Type chess or select the BlueJ icon named chess in the list of files. **Do not use any other name for your jar file** (e.g., lab7, my_project, etc.).

   2.3. Click Create. BlueJ will create a file named chess that has extension .jar. (Note: you don't type this extension when you specify the filename in Step 2.2; instead, it's automatically appended when the jar file is created.) The jar file will contain copies of the Java source code and several other files associated with the project. (The original files in your chess folder will not be removed).

3. Before you leave the lab, log in to cuLearn and submit chess.jar. To do this:

   3.1. Click the Submit Lab 7 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag chess.jar to the File submissions box. Do not submit another type of file (e.g., a .java file, a RAR file, a .txt file, etc.)

   3.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 3.4. If you instead want to replace or delete your "draft" file submission, follow the instructions in Step 3.3.

3.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

   3.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

   3.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

3.4. Once you're sure that you don't want to make any changes to the project you're submitting, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".

**Extra Practice**

Read about the other pieces used in the game of chess (http://en.wikipedia.org/wiki/Chess_piece). Define subclasses of `ChessPiece` that model the king, queen, knight and pawn. (Wikipedia has a page for each piece that describes the legal moves for that piece.)