Guessing Game with Objects

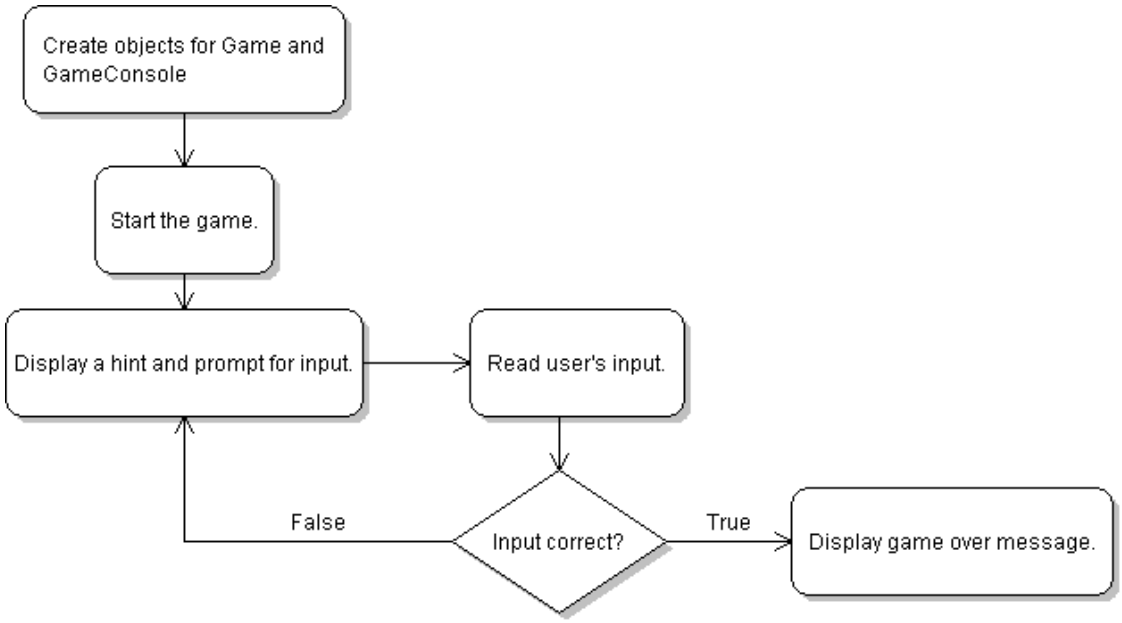
Objectives	1. Practice designing and implementing an object-oriented program. 2. Use Console I/O in Java.
Tasks	1. Design the program (problem 1). 2. Implement the program and write documentation.. Test it often! 3. Submit your code to Github as project named **GuessingGame**.

Problem 1: Guessing Game Design

Design a guessing game that is played on a console, like this:

```
I'm thinking of a number between 1 and 20.
What is your guess?   8
Sorry, too small.
What is your guess?  14
Sorry, too large.
What is your guess?  11
Right! The secret is 11.
```

The sequence of activities in the game is:



Game Design Using Objects

In object-oriented programs, objects have *responsibilities*. Try to design objects so that:

(a) objects behave like *real things* in the problem you are trying to solve

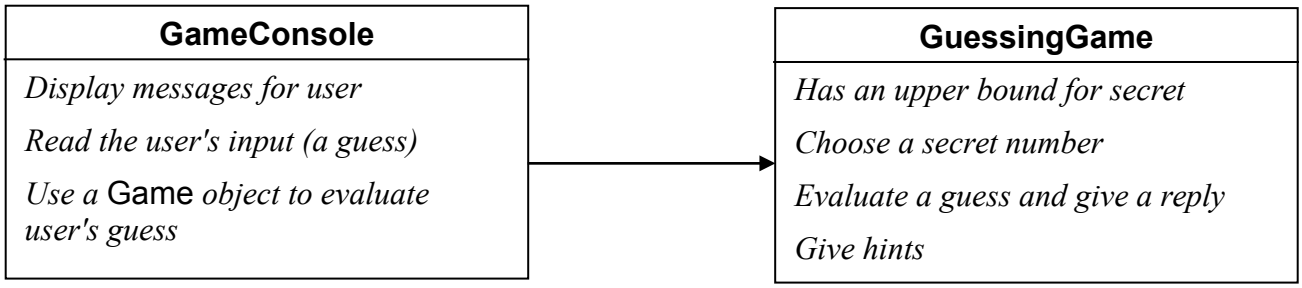
(b) each object has only one set of closely related responsibilities.

Try to make your objects simple and cohesive. This makes your program easier to understand, easier to maintain, and easier to modify.

In the guessing game, there are 3 sets of **responsibilities**:

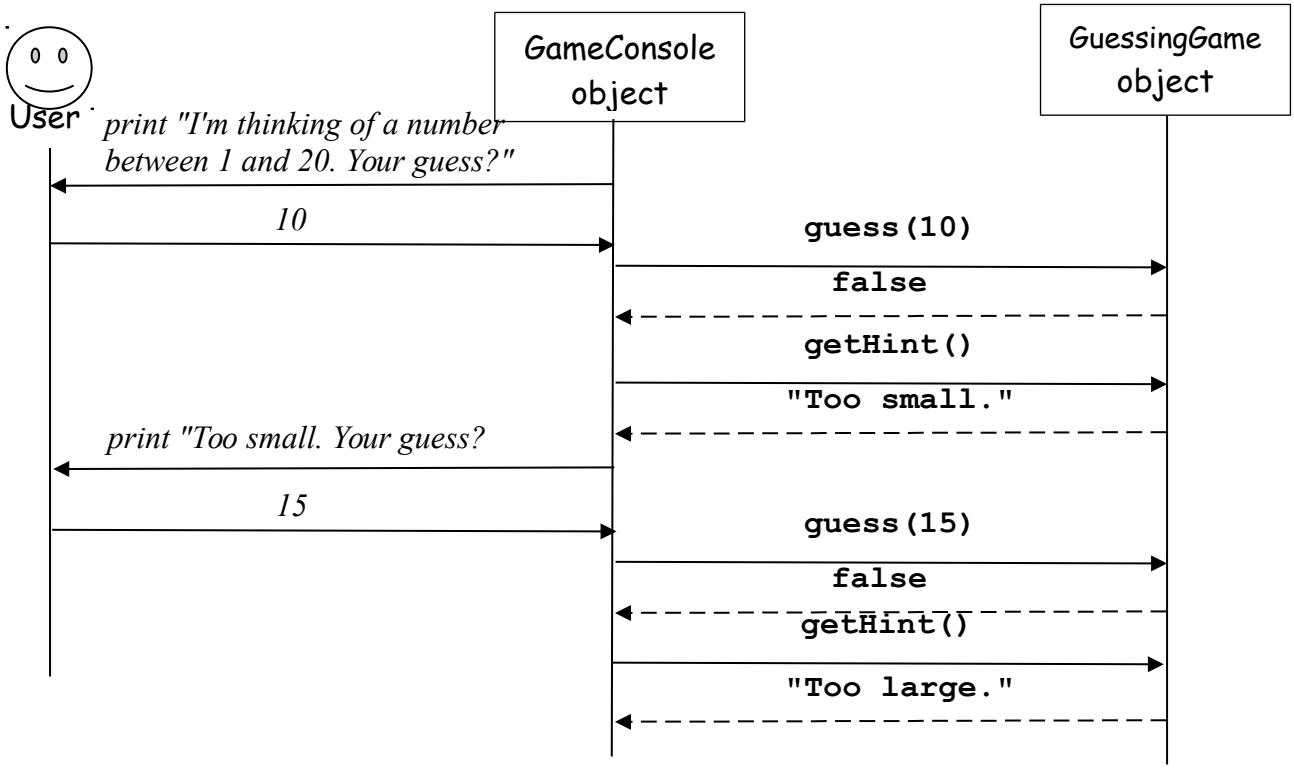
1. *Interact with the user: print messages and read input*
2. *Manage the game: know the secret number, evaluate a guess, create hint.*
3. *Create objects and start the game.*

We will define one class (a blueprint for a *kind* of object) for these two sets of responsibilities:



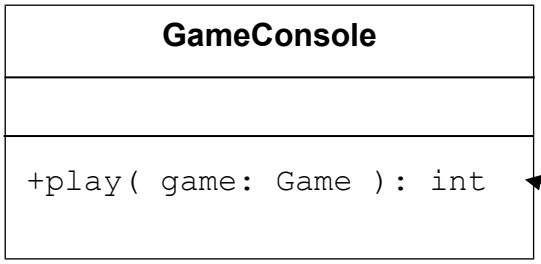
The arrow shows that the GameConsole needs to "know" about a Game object, because GameConsole has to "ask" the Game if a user's guess is correct, and "ask" for a hint.

The GameConsole and GuessingGame objects work together when you run the program. Here is an ***example***:



Application Design in UML

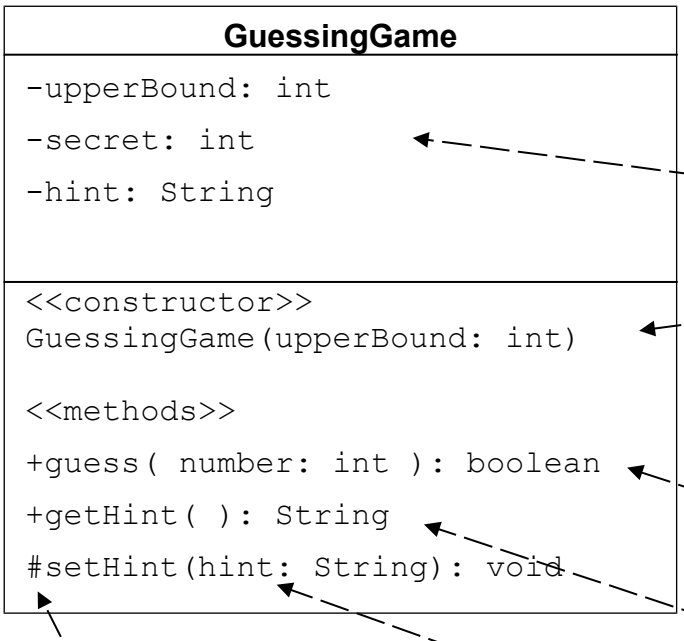
This is a UML class diagram for the two classes in our program.



The GameConsole class interacts with the user.

Responsibilities: performs input and output

play a game. The parameter is a Game to play. The **return value** is the solution.



The Guessing Game.

Responsibilities: knows about the game.

UML for attributes. "-" means private.
In Java write: `private int secret;`

constructor:
Choose a secret number.
Initialize *all* the attributes.

guess(number) evaluate the user's guess and return true or false. Also prepare a hint.

getHint( ) return the current hint.

setHint( ) is used only by the Game or a subclass of GuessingGame.

means a **`protected`** method. It is protected so we can *override* it in a future version of the game.

Problem 2: Write the GuessingGame Class and Test It

1. Create a new class named "Game" and open it with BlueJ editor (double click on class icon).

```
import java.util.Random;                // for random numbers
/**
 * Game of guessing a secret number.-- write a description here
 * @author Your Name                  -- your name, no parenthesis.
 */
public class GuessingGame {
    /* properties of a guessing game */
   //TODO Declare variables for attributes of the game
```

You can copy and paste code from the SampleGame class into the Game class to get started.

2. Define the attributes of Game. There should be 3 attributes -- as in the class diagram. For example:

```
private int upperBound;
```

3. Write the **constructor**. The constructor initializes a new object of the Game class. Do this:

```
/**
 * Initialize a new game.
 * @param upperbound is the max value for the secret number (>1).
 */
public GuessingGame( int upperbound ) { // this is a constructor
```

(a) set the **upperBound** for the secret number

(b) create a **secret** number by calling getRandomNumber (see next item)

(c) initialize the **hint**. Use: "**I'm thinking of a number between 1 and XX**".

```
}
```

4. Write a **method** to choose a random number between 1 and the upper bound. You need to import the java.util.Random class for this.

This is a separate method to make our code simpler and easier to read.

```
/**
 * Create a random number between 1 and limit.
 * @param limit is the upper limit for random number
 * @return a random number between 1 and limit (inclusive)
 */
private int getRandomNumber(int limit) {
    long seed = System.currentTimeMillis( );
    Random rand = new Random( seed );
    return random.nextInt(limit) + 1;

}
```

TODO

//TODO comment means something you have to do!

After you do it, delete the //TODO comment.

5. Write the guess(number) method that evaluates a guess. It also creates a hint (a String) which is saved for later.

```
public boolean guess( number )  {
  test whether the user's guess matches the secret.
  If correct, set hint to "Correct. The secret is XX.".
```

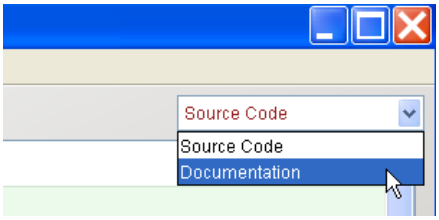
Otherwise, return false and set hint to "Sorry, your guess is too small." or "Sorry, you're guess is too large."
Use the setHint( ) method to set the hint.

6. Create an *accessor method* **`getHint( )`** that returns the **`hint`** attribute (**`hint`** variable).

In Java, an *accessor method* begins with the word "get" (or "is" for boolean values).

```
/**
 * Return a hint based on the most recent guess.
 * @return hint based on most recent guess
 */
public String getHint( ) {
    //TODO complete this.  Its easy!
```

7. Compile your code and fix errors.
8. View your Javadoc in the editor.
   Is it complete? Do you have @author, @param, @return tags?



9. Write a main method to test this class.
   a) does it create random numbers between 1 and upperbound? (not 0)
   b) does it give the correct hint for every guess?
   c) what if you guess something ridiculous (negative number or greater than upperbound)?

You should test every class! Don't assume that it is correct.

NO JAVADOC = NO CREDIT

If you don't write *good* Javadoc documentation for *all* your code, including class and method descriptions, you won't get *any credit* for your work.

Problem 3: Write the GameConsole Class and Test It

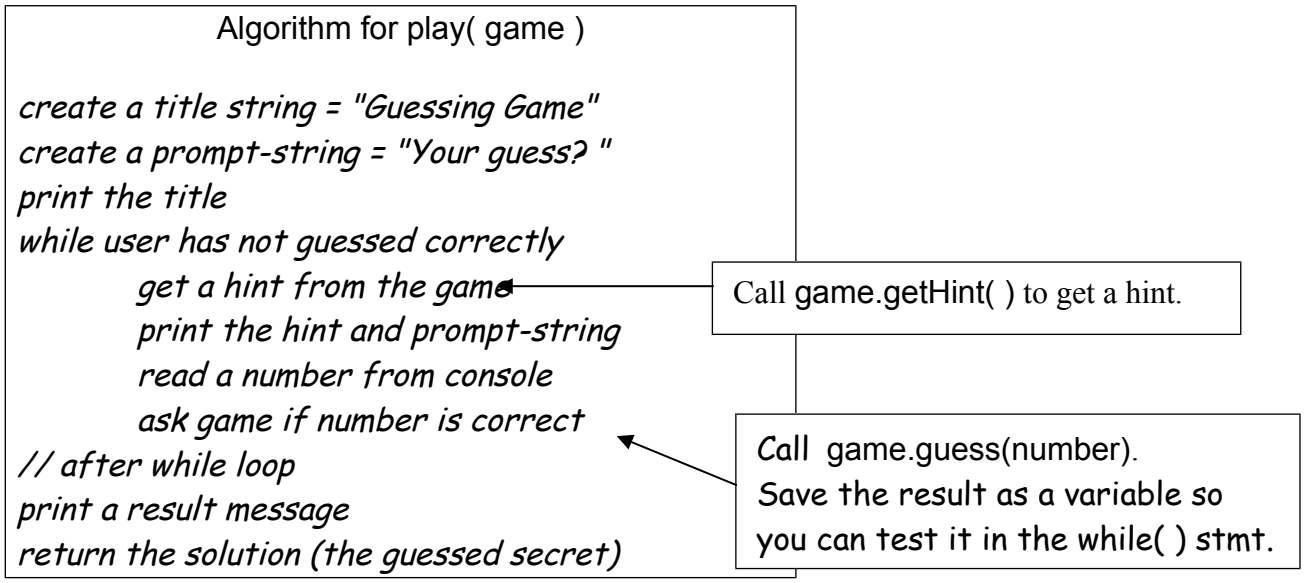
This class has only 1 method: play.

1. Create a new class named GameConsole.

2. Write the play(game) method.

```
public int play(Game game) {
    //TODO play game on the console.  Return the secret number when user guesses it.
}
```

The algorithm for play(game) is like this:



4. **Write Javadoc** comments for the GameConsole class and the play() method.

```
/**
 * The play method plays a game using input from a user.
 * @param ...
 * @return ...
 */
```

Problem 4: Write a Main class to create objects and start the game

1. Create a class named Main with a static **main** method.

> The job of the main method is to create objects, connect user interface to the game, and start the user interface. main is a **static** method. main should be *simple (no application logic)*.

```
/** create objects and start the game */
public static void main( String [] args ) {
    GuessingGame game = new GuessingGame( upperbound );
    GameConsole ui = new GameConsole( );
    ui.play( game );
}
```

5. Compile your GameConsole until there are no errors.

6. Run it as demonstrated in class (right click on GameConsole and select main).

Problem 5: Count the guesses

1. Add a counter to the GuessingGame class that counts how many guesses the user makes.

Add a counter *only* to the GuessingGame class.

2. Provide an *accessor* method for the counter. *Accessor* is a "**`getCount`**( )" method.
3. Write good Javadoc for the *accessor* method including @returns ... Test the code.
4. When the user guesses the correct answer, the game should tell him how many guesses he used:

```
Correct.  You used 7 guesses.
```

Problem 6: (Optional) Use Dialog Boxes instead of console input

This problem shows the benefit of using separate objects for separate responsibilities.

Instead of playing on the console, we can use dialog boxes. We won't need to rewrite the application! Just *replace* the `GameConsole` class with a new `GameDialog` class. We can *reuse* the **`Game`** class.

In a real project, you will often write a simple console input class to test your program logic, then write a graphical interface for really playing the game.

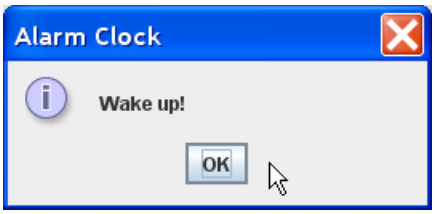
1. Create a new class named GameDialog. Copy the methods from GameConsole to GameDialog.

2. Modify GameDialog to use **dialog boxes** instead of **`System.in`** and **`System.out`**.

How to Use Dialog Boxes

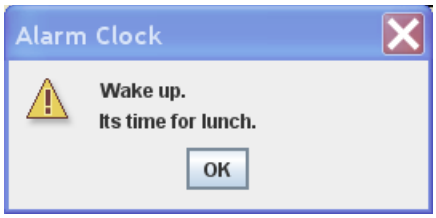
JOptionPane (in package javax.swing) contains many useful dialogs. Here are some examples. You can run these examples interactively in BlueJ.

Message Dialog




```
import javax.swing.JOptionPane;
String title = "Alarm Clock";
String message = "Wake Up.";
int type = JOptionPane.INFORMATION_MESSAGE;
JOptionPane.showMessageDialog( null, message, title, type);
```

Message Dialog with multi-line message:




```
message = "Wake up.\nIts time for lunch";
type = JOptionPane.WARNING_MESSAGE;

JOptionPane.showMessageDialog( null, message, title, type);
```

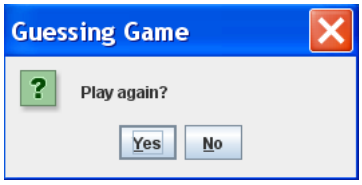
Input Dialog:

Use this to print a hint and ask user to guess a number.


```
title = "OOP Coffee Shop";
message = "We have all kinds of coffee.\n";
message += "What would you like to drink?";
type = JOptionPane.QUESTION_MESSAGE;
String reply = JOptionPane.showInputDialog(null,message,title,type);
if ( reply == null )
     /* user pressed Cancel */;
else orderCoffee( reply );
```

Test if user presses Cancel *The dialog returns a* `null`.

Confirm Dialog:




```
String title = "Guessing Game";
message = "Play again?";
type = JOptionPane.YES_NO_OPTION;
int opt = JOptionPane.showConfirmDialog(null, message, title, type);
if ( opt == JOptionPane.YES_OPTION ) /* user pressed "Yes" */;
```