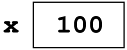
Introduction to Object-Oriented Programming (OOP)

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

References and Reference Types

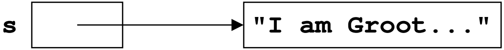
- Recall that Java has **primitive types** and **reference types**
  - For a primitive type, a data value is stored directly in the memory location associated with the variable

```
int x = 100;
```



  - For a reference type, the stored value is a **reference** to an **object** that is stored else where

```
String s = "I am Groot...";
```



References?

- What do we mean by **references**?
  - Data stored in a reference variable is just an address (location) where an object is stored
  - So, imagine that I have a contact book
    - Page 5 contains my good friend Joe and his address
    - At Joe's physical address, it contains Joe's house (object)
    - With Joe's address, I can contact him by snail mail (USPS)
    - I can go to visit Joe if I want to
    - I can modify the object at Joe's address (his house) (e.g., paint Joe's house)
    - If I erase Joe's address from my contact book, I can no longer access Joe's house. However, Joe's house is still there.

Classes and Objects

- What about **Objects**?
- Let think about a physical object (e.g., a smartphone)
  - You need to design how to build it first
  - A single design (plan/blueprint) can be used to build tons of the same smartphone
    - You cannot use a plan to text a friend
  - If you have access to a smartphone (pyysical object), you can
    - call or text, play games, watch movies, and surf the Web.
  - Data (e.g., musics, pictures, contacts) are stored in the phone's memory
    - Unfortunately, we cannot access data directly
    - We can only do what the phone (os) allows us to do

Classes

- **Classes** are **blueprints** of our data
  - Plans to construct objects
  - The class structure provides a good way to **encapsulate** the **data** and **operations** of a new type all together
    - **Instance data** and **instance methods** are fundamental feature of OOP
    - The **data** gives us the structure of the objects
    - The **operations/methods** show us how to use them
- Example: The class `String`

Classes and Objects

- User of the class **know the general nature of the data**, and the **public methods** but **NOT** the implementation detail
  - You do not need to know how a phone is manufacturer
  - Just need to know how to use it
  - Example: `BigInteger`
- We call this **data abstraction**
  - Compare to **functional abstraction** of methods discussed earlier
- Java classes determine the structure and behavior of Java object
  - Again, a blueprint is just a plan
- **Java objects** are **instances** of **Java classes**

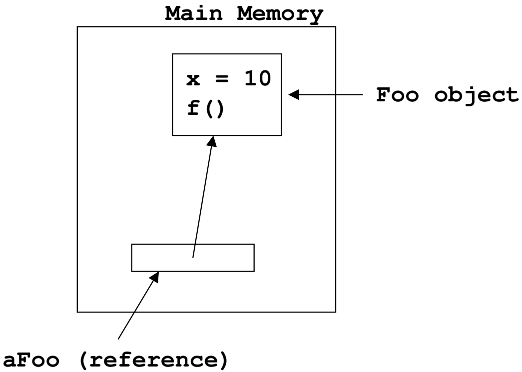
Classes and Objects

```
class Foo
{
   int x;
   void f();
   ...
}
```

Definition of the class Foo

Declaring Foo variable
Creating Foo obejct

```
Foo aFoo;
aFoo = new Foo(10);
```



More about References

- Let's now see some of the **implications of reference variables**
  - **Declaring a variable does not create an object**

    ```
    StringBuilder s1, s2;
    ```

    - We have no actual `StringBuilder` objects. Just two variables that could access (refer to) them
  - We must create (construct) objects separately from declaring variables
    - To construct objects, we must use the `new` operator or call a method that will create an object for us

      ```
      s1 = new StringBuilder("Hello");
      ```

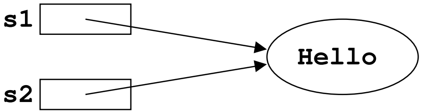
    - `s1` new references an **instance** of a **StringBuilder object** but `s2` does not

More about References

- What value does `s2` have?
  - For now, we will say that we should not expect that `s2` will have a value
  - We must initialize it before we can use it
  - If we try to use it without initializing, we will get an error
- Multiple variables can access (refer) and alter the same object

```
s2 = s1;  // make s2 to have the same value as s1
```

  - Since `s1` contains the reference (memory address) of the `StringBuilder` object constructed earlier, now `s2` contains the same reference.
  - Any change via `s1` or `s2` will update the same object

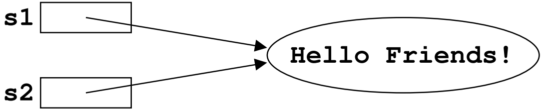


More about References

- **Properties** of objects (public methods and public instance variables) are access via the **dot** notation

```
s1.append(" Friends!");
```

  - The process of accessing an object in this way is called **dereferencing** that object
    - `s1` is a reference (address)
    - `s1.<whatever>` means:
      Go to the object whose address is stored in `s1`
      Access/Call the specified variable or method



    - Note that in this case `s2` will also access the appended object

Comparing Objects

- Comparison of reference variables compares the reference. **Not the object**

```
StringBuilder s3 = new StringBuilder("Hello Friends!");
if(s1 == s2)
    System.out.println("Equal");    // print Equal
if(s1 == s3)
    System.out.println("Equal");    // Does not print
```

  - Recall that `s1`, `s2`, and `s3` are variables storing addresses
    - The == simply compares those address values
    - In other words, are they referencing the same location
  - What if we want to compare the objects?
    - Do the objects (where ever they are) have the same data stored within them?

Comparing Objects

- We use the `equals()` method
  - This is generally defined for many Java classes to compare data within objects
  - We will see how to define it for our own classes later
  - Unfortunately, the `equals()` methods is not **redefined** for the `StringBuilder` class, so we need to convert our `StringBuilder` objects into `String` objects in order to compare them:

```
if(s1.toString().equals(s3.toString()))
    System.out.println("Same");
```

  - We will also use the `compareTo()` method later
- It seems complicated but it will make more sense when we get into defining new classes
- Again:
  - The == operator shows use that it is the same object
  - The `equals()` method show us that the values are in some way the same (depending on how it is defined)

The `null` Reference

- References can be set to `null` to initialize or reinitialize a variable
  - The `null` references cannot be accessed via the **dot** notation
  - Run-time error will occur

```
s1 = null;
s1.append("This will cause a run-time error");
```

- Why?
  - The method calls are associated with the **object** that is being accessed. **Not** with the variable
  - If there is no object, there are no methods available to call
  - Results in `NullPointerException`, a common error. So, remember it
- Let's take a look at `ex8.java`

- Object-Oriented Programming (OOP) consists of 3 primary ideas:
  - **Encapsulation** and **Data Abstraction**
    - Operations on the data are considered to be part of the data type
    - We can understand and use a data type without knowing all of its implementation details

      No need to know how the data is represented
      No need to know how the operations are implemented
      Just need to know the interface or method headers
      (how to **communicate** with the object)

- **Inheritance**
  - Properties of a data type can be passed down to a sub-type — we can build new types from old ones
  - We can build class hierarchies with many levels of inheritance
  - More in chapter 11
- **Polymorphism**
  - Operations used with a variable are based on the class of the object being accessed, not the class of the variable
  - Parent type and sub-type objects can be accessed in a consistent way
  - Again, more in Chapter 11

Encapsulation and Data Abstraction

- Consider primitive types
  - Each variable represents a single **simple** data value
  - Any operations that we perform on the data are external to that data
  - Example: `x + y`
- Now consider the **data**
  - In many application, data is more complicated that just a simple value
  - Consider a **Polygon** – a sequence of connected dots
  - Data of a `Polygon` are:
    - `int[] xpoints` – an array of x-coordinate
    - `int[] ypoints` – an array of y-coordinate
    - `int npoints` – the number of points actually in the polygon
  - Note that an individual data is just an `int` but all together they make up a `Polygon`
  - This is fundamental to OOP

Encapsulation and Data Abstraction

- Consider the **operations**
- What a `Polygon` can do?
  - We are seeing what a `Polygon` **can do** rather tan what can be done with it
  - This is another fundamental idea of OOP – object are **active** rather than passive
  - Examples:
    - `void addPoint(int x, int y)`: Add a new point to a `Polygon`
    - `boolean contains(double x, double y)`: is point `(x, y)` within the boundaries of the `Polygon`
    - `void translate(int deltaX, int deltaY)`: move all points in the `Polygon` by `deltaX` and `deltaY`

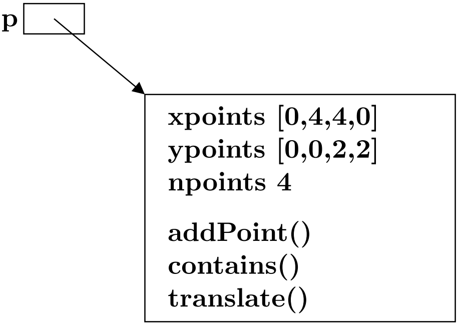
- These operations are actually (logically) part of the polygon itself

```
int[] xs = {0, 4, 4};
int[] ys = {0, 0, 2};
int num = 3;
Polygon p = new Polygon(xs, ys, num);
p.addPoint(0, 2);
if(p.contains(2, 1))
    System.out.println("Inside p");
else
    System.out.println("Outside p");
p.translate(2, 3);
```

- We are not passing the `Polygon` as an argument, we are calling the methods **from** the `Polygon`

Encapsulation and Data Abstraction

- Objects enable us to **combine the data and operations** of a type together into a single entity (**encapsulation**)

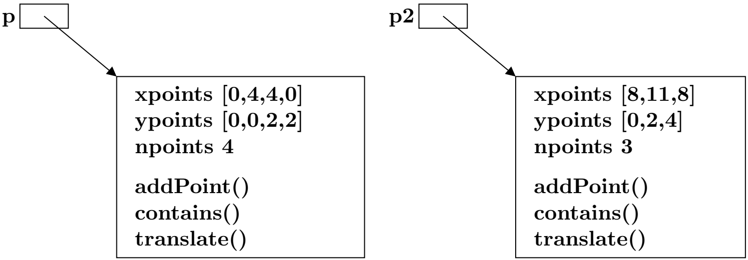


- Operations are always implicitly acting on the object's data
  - Example: translate means translate the points that make up `p`

Encapsulation and Data Abstraction

- For multiple objects of the same class, the operations act on the object specified

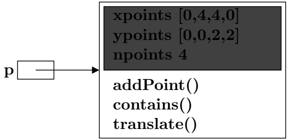
```
int[] anotherXs = {8, 11, 8};
int[] anotherYs = {0, 2, 4};
Polygon p2 = new Polygon(anotherXs, anotherYs, 3);
```



Encapsulation and Data Abstraction

- We do not need to know the implementation details of a data type in order to use it
  - This includes the methods **and** the actual data representation of the object
- This concept is exemplified through objects
  - We can think of an object as a container with data and operations inside (i.e., **encapsulating** them)
    - We can see some of the data and some of the operation but others are kept hidden from us
    - The ones we can see give us the functionality of the objects

- We can use `Polygon` as long as we know the method names, parameters,and how to use them
  - We do not need to know how the actual data is stored
  - We do not need to know how methods are implemented
- For a `Polygon`, we know is a sequence of points
  - How a point is store?
  - How a sequence of points are stored?
  - Does it maintain the number of points?
  - How the methods are implemented?
- We can just use a `Polygon` without knowing its detail implementation
  - **Data Abstraction**



Instance Variables

- Let's look at the `AbstractStringBuilder` class which is inherited by the `StringBuilder` class (comments are removed):

```
abstract class AbstractStringBuilder implements
               Appendable, CharSequence
{
    char[] value;
    int count;
    :
}
```

- **Instance Variables**
  - These are the **data** values **within an object**
    - Used to store the object's information after constructed
  - As mentioned earlier, when using data abstraction, **we do not need to know explicitly what these are** in order to use a class
  - Let's look at the API for the ▸ StringBuilder
    - The instance variables are not even shown here

Instance Variables

- Again, the `AbstractStringBuilder` class:

```
abstract class AbstractStringBuilder ...
    char[] value;
    int count;
```

  - It is a variable-length array with a counter to keep track of how many locations are being used and is actually inherited by the `StringBuilder` class
- Many instance variables are declared with the keyword `private`
  - They **cannot be directly accessed outside the class itself**
  - Base on the data abstraction we discussed earlier
    - We do not need to know how the data is represented in order to use the type
    - Need need to let us see it
  - Note that there is no keyword `private` shown above
    - Private to the package by default

Encapsulation and Data Abstraction

- Again, ideas of **encapsulation** and **data abstraction**
  - **Encapsulation** allows the instance variables to be separated or hidden from the user of a class
    - **Private** declarations and (we will see later) **protected** declarations are not directly accessible by the user of a class
  - **Data abstraction** enables a user to not require direct knowledge of these variables in order to use a class

Class Methods vs Instance Methods

- Recall that methods we discussed earlier were called **class methods** (or **static methods**)
  - These were not associated with any object
- Now, however in this case, we will associate methods with object (as shown with Polygon)
  - Each object has its own set of methods
- These methods are called **instance methods** because they are associated with individual **instances** (or objects) of a class
  - There are the **operations** within an object

```
StringBuilder b = new StringBuilder("Luke.");
b.append("I am your FATHER!!!");
System.out.println(b.toString());
```

  - Need `b.append(...);` instead of just `append(...);`

Class Methods vs Instance Methods

- **Class methods** have no implicit data to act on
  - All data must be passed into them using arguments
  - Class methods are called using

    `ClassName.methodName(parameter list)`
- **Instance methods** have implicit data associated with an object to act on
  - Other data can be passed as arguments, but there is always an underlying object to act upon
    - This is because they are encapsulated within that same object
  - Instance methods are called using:

    `variableName.methodName(parameter list)`

    where `variableName` is a reference to an object

Encapsulation and Abstraction Summary

- In summary:
  - **Objects** allow us to **encapsulate data** and **operations** together into a single entity
    - **Instance variables** define the **data** within the object
    - **Instance methods** define the **operations** to be used by the object
  - The **instance variables** and **instance methods** are specified in the class definition
  - **Objects** are instances of class which contain the specified data and methods

Encapsulation and Abstraction Summary

- Because of **data abstraction**
  - To use objects in our program, **we only need to know**:
    - The general idea of the data to be store
    - What the instance methods are (i.e. names)
    - What they are suppose to do (i.e. general function)
    - What parameters they need
  - We do not need to know
    - The specific instance variables (names or types)
    - The instance methods implementations
- **Encapsulation** and **data abstraction** are closely related
  - Encapsulating the data and methods in an object enables programmer to **restrict access** and require abstraction for use

Constructors, Accessors, and Mutators

- Instance methods can be categorized by what they are designed to do:
- **Constructor**:
  - These are **special instance methods that are called when an object is first created**
  - They are the only methods that **do not have a return value**. Not even `void`
  - They are typically used to initialize the instance variables of an object

```
StringBuilder b1 = new StringBuilder("I am Groot!!!");
StringBuilder b2 = new StringBuilder();   // default
                                          // constructor
StringBuilder b2 = new StringBuilder(10); // set capacity
                                          // to 10
```

- **Accessors** (or **getters**):
  - These methods are used to **access the object in some way without changing it**
  - Usually used to get information from it
  - No special syntax - categorized simply by their effect

```
StringBuilder b = new StringBuilder("Hello World!!!");
char c = b.charAt(4);          // c is now 'o'
String s = b.substring(3, 9); // s is now "lo Wor"
           // note that end index is NOT inclusive
int n = b.length();            // n is now 14
```

  - These methods give us information about the object `b` (`StringBuilder`) without revealing the implementation details

- **Mutators** (or **setters**):
  - Used to **change the object in some way**
  - Since the instance variables are usually **private**, we use mutators to change the object in a specified way without needing to know the instance variables

```
b.setCharAt(0, 'Y');    // b is now "Yello World!!!"
b.delete(6, 7);         // b is now "Yello orld!!!"
b.insert(5, "w -");    // b is now "Yellow - orld!!!"
```

  - These method change the contents or properties of the `StringBuilder` object
- We use accessors and mutators to **indirectly access the data**, since we do not have direct access
- See `ex9.java`

Simple Class Example

- We can use these ideas to write our own class
- Let's look at a very simple example
  - A **circle** is a two-dimensional object with radius
- Let's look at the `Circle` class

```
public class Circle
{
    :
}
```

  - **Instance variable**

    ```
    private double radius;
    ```

    Cannot directly access it from outside of this class
  - **Constructor** takes a real number (`double`) as the argument and initialize a new circle with the given radius

    ```
    public Circle(double aRadius)
    {
        radius = aRadius;
    }
    ```

    - No return type (not even `void`)
    - Exact same name as the class name

Simple Class Example

- The `Circle` class (so far)

```
public class Circle
{
    private double radius;

    public Circle(double aRadius)
    {
        radius = aRadius;
    }
}
```

  - **Accessors**: Get information about the object of this class

```
public double area() {...}
public double circumference() {...}
public String toString() {...}
```

  - **Mutator**: Change something about the object of this class

```
public void setRadius(double aNewRadius) {...}
```

Simple Class Example

- The `Circle` class

```
public class Circle
{
    private double area;

    public Circle(double aRadius) {
        radius = aRadius;
    }
    public double area() {
        return Math.PI * radius * radius;
    }
    public double circumference() {
        return 2 * Math.PI * radius;
    }
    public String toString() {
        return "Circle with radius " + radius;
    }
    public void setRadius(double aNewRadius) {
        radius = aNewRadius;
    }
}
```

More on Classes and Objects

- **Classes**
  - Blueprints
  - Define the nature and properties of objects
  - Example:

```
public class MyClass {...}
```

- **Objects**
  - Instance of classes
  - Needed to be constructed

```
MyClass mc = new MyClass(...);
```

- Let's learn more about these by developing another example together
- Goal:
  - Write a **class that represents a playlist???** (group of songs)
  - Write a simple driver program to test it

Developing Another Example

- Remember the things we need for a class:
  - **Instance variables**
    - Fill in ideas from board
  - **Constructors**
    - Fill in ideas from board
  - **Accessors**
    - Fill in ideas from board
  - **Mutators**
    - Fill in ideas from board

Developing Another Example

- Once we have the basic structure of the class, we can start writing/testing it
- A good approach is to do it in a modular (step-by-step) way
  - Example: Determine some instance variables, a constructor (or two), and an accessor to **output** data in the class
  - Write a simple driver program to test these features
    - Once a method has been written and tested, we do not have to worry about it anymore
  - Add more to the class, testing it with additional statements in the driver program

Developing Another Example

- There are formal approaches to doing this
  - **Unit Testing**:
    - A framework/program is developed to test the required functionalities of the class (or "unit") in a formalized way
    - Test to make sure the class behaves the way it is supposed to behave
    - See `https://en.wikipedia.org/wiki/Unit_testing`
  - **Java Assertions**:
    - Conditions that should always be true (e.g., `currentCount <= maxCount`)
    - If an assertion becomes false, an `AssertionError` is thrown
    - See `https://docs.oracle.com/javase/7/docs/technotes/guides/la`

Introduction to Java Files

- So far, our program have:
  - read input from the keyboard
  - written output to the console (monitor)
- It is fine in some situations, but is not so good in others:
  - What if we have a large amount of output that we need to save?
  - What if we need to initialize a database that is used in our program?
  - What if output from one program must be input to another?
- In these situations, we need to use **files**

Introduction to Java Files

- Most files can be classified into two groups:
  - Text Files (will be discussed now)
  - Binary Files (will be discussed later)
- A Text file is simply a sequence of ASCII characters stored sequentially
- Any "larger" data types are still stored as characters and must be "built" when they are read in
  - Example: Strings are sequences of characters
  - Example: `int` (integers) are also sequences of characters, but interpreted in a different way

- Example: `int`
  - To create an actual `int` **we need to convert the characters into an integer**
    - This is what `nextInt()` of the `Scanner` class does
    - We will discuss the conversion procedure more later
  - If we want to read data into an object with many instance variables, we can read each data value from the file, then assign the object via a constructor or via mutators
    - See `PlayListTest.java`
  - If we want to fill an array, we can read in as many values as we need
    - We may first need to read in how many values there are, then create the array and read in the actual data
    - See `PlayListTest.java` and another example soon

Introduction to Java Files

- Similarly, if we have data in our program that we wish to save to a text file, we need to first convert it into a sequence of character (i.e. a `String`)
  - Example: The `toString()` method for a class
- However, now we need a different class that has the ability to write data to a file
  - There are several classes in Java that have this ability
  - For now, we will focus on the `PrintWriter` class
    - An object of the `PrintWrite` class allows us to write primitive types and Strings to a text file
    - See PrintWriter API

- The `PrintWriter` is fairly simple to use
    - See `FileTest.java`
- However, when creating the file, an **Exception** can occur
    - We will see how to handle this later
    - For now, we will simply "pass the buck"
    - We do this via the "throws" clause in the **method header**
        - States that we are not handling the exception
        - Must be stated in a method where the exception could occur or in any method that calls a method (since the exception is passed on)
    - See `FileTest.java`

- Step-by-step example with APIs

```
import java.io.*;    // The File class
public class MyFile {
    public static void main(String[] args) {
        File inputFile = new File("aTextFile.txt");
    }
}
```

  - The File API

```
public File(String pathname)

Creates a new File instance by converting the given pathname string into an
abstract pathname. If the given string is the empty string, then the result is
the empty abstract pathname.

Parameters:
    pathname - A pathname string
Throws:
    NullPointerException - If the pathname argument is null
```

    - `NullPointerException` doe not need the "throws" clause
    - Check RuntimeException API

Introduction to Java Files

- Step-by-step example with APIs

```
1 import java.io.*;    // The File class
2 import java.util.*;  // The Scanner class
3 public class MyFile {
4     public static void main(String[] args) {
5         File inputFile = new File("aTextFile.txt");
6         Scanner inputFileScanner = new Scanner(inputFile);
7     }
8 }
```

  - The Scanner API

```
public Scanner(File source)
        throws FileNotFoundException
Constructs a new Scanner that produces values scanned from the specified file.
Bytes from the file are converted into characters using the underlying
platform's default charset.
Parameters:
    source - A file to be scanned
Throws:
    FileNotFoundException - if source is not found
```

  - Without the “throws” clause, we will get

```
MyFile.java:6: error: unreported exception FileNotFoundException; must be caught
or declared to be thrown
Scanner inputFileScanner = new Scanner(inputFile);
                           ^
1 error
```

Introduction to Java Files

- Step-by-step example with APIs

```
import java.io.*;    // The File class
import java.util.*;  // The Scanner class
public class MyFile throws IOException {
    public static void main(String[] args) {
        File inputFile = new File("aTextFile.txt");
        Scanner inputFileScanner = new Scanner(inputFile);
    }
}
```

- Note that the error was about `FileNotFoundException`
  - The `FileNotFoundException` is a class
  - It is a **subclass** of the class `IOException`
    - More about **subclass** and **superclass** later

Introduction to Java Files

- Let's read all lines from the file

```java
import java.util.*; // The Scanner class
import java.io.*; // The File class
public class MyFile {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("aTextFile.txt");
        Scanner inputFileScanner = new Scanner(inputFile);
        while(inputFileScanner.hasNextLine()) {
            System.out.println(inputFileScanner.nextLine());
        }
        inputFileScanner.close();
    }
}
```

- A file should be closed after we finish reading or writing
    - Free up resources
    - Data generally are written to a buffer (faster) before writing to a file (slower)
        - By closing the file, data in the buffer will be pushed to the file

Introduction to Java Files

- Let's add the PrintWriter class

```
import java.util.*; // The Scanner class
import java.io.*; // The File class
public class MyFile {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("aTextFile.txt");
        Scanner inputFileScanner = new Scanner(inputFile);
        PrintWriter outputFileWriter =
                    new PrintWriter(new File("output.txt"));
        while(inputFileScanner.hasNextLine()) {
            System.out.println(inputFileScanner.nextLine());
        }
        inputFileScanner.close();
    }
}
```

- The Scanner API

```
public PrintWriter(File file)
          throws FileNotFoundException
:
Throws:
   FileNotFoundException - ...
   SecurityException - ...
```

  - SecurityException is a subclass of RuntimeException

Introduction to Java Files

- Read from one write to the other

```
import java.util.*; // The Scanner class
import java.io.*; // The File class
public class MyFile {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("aTextFile.txt");
        Scanner inputFileScanner = new Scanner(inputFile);
        PrintWriter outputFileWriter =
                      new PrintWriter(new File("output.txt"));
        while(inputFileScanner.hasNextLine()) {
            String aLine = inputFileScanner.nextLine();
            outputFileWriter.println(aLine);
        }
        inputFileScanner.close();
        outputFileWriter.close();
    }
}
```

  - The `println()` method of the `PrintWriter` prints to file instead of the console screen
  - Again, do not forget to close the file when done

- ```
  % This is a comment
  First verbatim line.
  Second verbatim line.
  Third verbatim line.
  ```
  ```