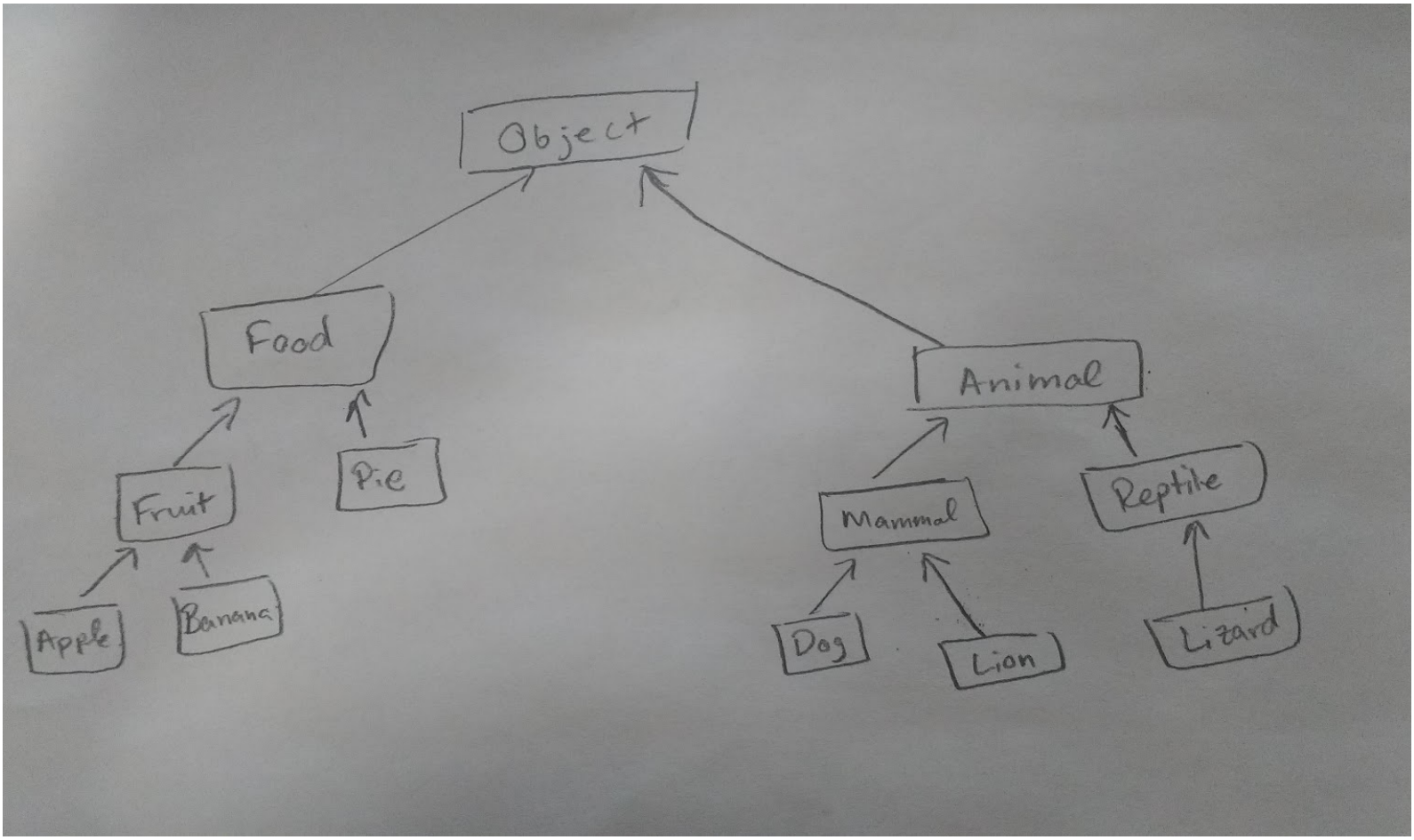
Inheritance:

1.



All classes extend Object directly or indirectly (by extending a class that extends Object), so Object should be at the top.

2. No, that doesn't really make sense. The point of extending a class is to gain or augment its behavior (e.g. override a method or provide new ones). A class already has access to its own properties, so a class extending itself wouldn't have a purpose.

3. A
C c = new C() constructs a C object. First java checks if there is a constructor in C. There isn't, so the default constructor runs. In constructing a C, the first step is to construct the B portion of the C. B also doesn't have a constructor, so its default constructor runs. Likewise, A's default constructor is called, which call's Object's no-arg constructor. Then, String s is constructed in the A portion of the B portion of the C object. Then, the rest of B object is constructed, which shadows String s. Finally, the rest of the C object is constructed, which shadows String s again. Notice that all three fields named `s` exist at the same time.

Next, c.foo(c.s) calls foo in C which calls super.foo. super.foo refers to foo in B since B is C's superclass. foo in B prints super.s. Since A is B's superclass, super.s is s in A which is "Class A".

Then, execution goes to the next line in foo in C which prints super.s. That refers to s in B which is "Class B".

Then execution goes to the next line in main which prints c.s. s in C is "Class C".

4. Constructing an object and setting it equal to a more general (higher on the inheritance tree) is allowed while setting an object to a less general (lower on the inheritance tree) is not allowed.

Given the above code, is each of the following declarations valid?

A. Animal a = new Mammal(); // YES! A Mammal is an Animal. Animal is higher on the inheritance tree than Mammal since Mammal extends Animal. Any attribute or method for an Animal also exists for a Mammal. For example, if Animal were to have an age attribute, then it makes sense for Mammal to also have an age attribute.
B. Bear d = new Mammal(); // NO! A Mammal may not be a Bear! If Bear were to have an attribute numFishEatenToday, Mammal wouldn't. Not all Mammals eat fish, so it doesn't make sense for Mammal to have an attribute about eating fish. For example sheep don't eat fish, but they are mammals.
C. Animal b = new Bear(); // YES! A Bear is an Animal. Animal is more general than Bear, so Animal is higher on the inheritance tree.
D. Mammal c = new Bear(); // YES! A Bear is a Mammal

5. At compile time, `a` is thought to be an Animal since `a` is of declared type Animal. At runtime, java realizes that `a` is actually a Mammal since a Mammal is what was actually constructed. Similarly, `b` is thought to be an Animal at compile time. At runtime, java realizes it is a Bear. At compile time, `c` is thought to be a Mammal. At runtime, it is found to be a Bear.

Given the above code, is each of the following method calls valid?
A. a.eat(); // OK. Mammal doesn't override eat(), so eat() refers to the eat() in Animal. Eat anything.
B. b.eat(); // OK. At compile time, b is thought to be an Animal, which has an eat method. So there isn't a compiler error. At runtime, java finds b is actually a Bear. Bear has an eat() method, so there is no runtime error. Since b is really a Bear, Bear's eat() is called. Eat fish.
C. c.hibernate(); // WRONG! Mammal doesn't have this method! So at compile time, java looks for a hibernate method in Mammal. There isn't one which leads to a compiler error. At runtime, java would have found that c is a Bear, and Bear has a hibernate method. Since this line leads to a compiler error, it never even gets to runtime.

6. C
A subclass is like a subcategory of its parents. For example, Student might be a subclass of Person since students are people (we think).
A. Wrong since it wouldn't make sense for Student to only get access to methods from Person. For example, if Person has a member variable height, so should Student. Since students are people, students have heights.
B. Wrong. If Person has a method speak(), then so should Student. Since students are people, students can speak too.
C. Correct. A subclass inherits all methods and fields from a superclass. Note that this even includes private members, even though the subclass won't be able to access them directly. In such a case, the subclass can still call methods on its superclass portion, which do have access to these private members.
D. Wrong. If only member variables are inherited, then this has the issue of option B. If only methods are inherited, then this has the issue of option A.

7. D
A. Wrong. Jolt *is-a* SoftDrink is the relationship between Jolt and SoftDrink, but `*is-a*` is not proper Java syntax.
B. Wrong. Implements is used for interfaces. The problem says that SoftDrink is a superclass, meaning it isn't an interface.
C. Wrong. `defines` isn't even a reserved word in Java.
D. Correct. `extends` is used in Java to create is-a and has-a relationships between classes.

8.
I’m an object
I’m throwable
I'm an exception

`e` is an instance of IllegalArgumentException. IllegalArgumentExceptions are Throwable as all exceptions extend Throwable. Instances of Throwable are also instances of Object as Throwable extends Object.
The instanceof operator checks if an object is an instance of a type of object. That is why instanceof returns true with IllegalArgumentException, Throwable, and Object.

9.
[A, B, C]
A
[A, B, C]
B
[A, B, C]
C

obj is an array of Objects containing 3 objects of types A, B, and C, respectively. The for loop goes through each object, o, in obj. So o is a reference to the A object in the 1st iteration, then a reference to the B object in the 2nd; and then a reference to the C object in the last iteration.
Calling Arrays.toString on obj calls toString on each element of obj. Every other line that prints is [A, B, C] because the toString of object A prints A, the toString of object B prints B, and the toString of object C prints C.
For a similar reason printing o prints A when o is an instance of A, B when o is an instance of B, and C when o is an instance of C.

10.
Huh, how about that!

a is an instance of A. b is an instance of B. c is initially a;i.e. c initial refers to the same instance of A to which `a`refers. Then c is set to an instance of C. It may seem like changing c from referencing an instance of A to referencing an instance of C shouldn’t be allowed, but since c is declared as type Object and both A and C implicitly extend Object, java will allow c to reference an instance of any type of Object (including A and C). a is then set to reference the same instance of C as c (since a is also declared as an Object, this is allowed).

C doesn’t override the equals method, so a.equals(c) uses Object’s equals method. Since a and c both reference the same instance of C, a.equals(c) returns true. Since B overrides the equals method, b.equals(a) calls B’s equals method. B’s equal method always returns true, so even though a and b are not the same object, b.equals(a) still returns true.
Note that a.equals(b) would return false since Object’s equals method would be used and a doesn’t equal b.

Exception Handling:

Note: Some of these are tricky. We encourage you to try out variations in Eclipse using the Eclipse debugger to check yourself!

#1-6. Since main[is printed right away (before any try-catch blocks) main[will always be printed.
Then the try block is entered, which calls methodA. So execution goes to methodA, where A[is printed on a newline. In the try block in methodA, methodB is called. In method B, B[is printed on a newline. From here each problem is different.

1. no exception is thrown

main[
A[
B[after C,after D,B-finally,]B
after B,]A
after A,after E,main-finally,]main

Starting from right after B[is printed, methodC is called. Since no execptions are thrown in this problem, we should assume that methodC doesn't do anything. So execution of methodB continues and prints "after C" on the same line. Entering the try block in methodB, methodD is called. Since methodD doesn't throw an exception in this problem, we should assume methodD doesn't do anything. So "after D" is printed on the same line. Since no exceptions where thrown, the catch blocks are skipped. Next the finally block is executed since finally blocks are always executed. "B-finally," is printed on the same line. Then]B is printed on the same line. Then a newline is printed.
Now execution returns to the try block in methodA. "after B," is printed. No exceptions were thrown, so the catch block is skipped and]A and a newline are printed.
Execution continues in the try block in main. "after A," is printed. Since no exceptions are thrown in this problem, we can assume methodE doesn't do anything. So "after E," is printed on the same line. The catch blocks are skipped, and the finally block is executed. "main-finally," followed by "]main" are printed on the same line.

2. methodC throws a GreenException?

main[
A[
B[main-green,main-finally,]main

Starting from right after B[is printed, methodC is called. methodC throws a green exception, so normal execution stops. Since methodC is not called in a try block in methodB, the exception cannot be caught in the catch blocks in methodB. So java looks in the method that called methodB (methodA). methodB was called from a try block, so java looks at the catch block in methodA to catch the GreenException. The catch block is for BlueException, so methodA cannot catch the GreenException. Since methodA was called from the try block in main, the catch blocks in main are the next place to look to catch the GreenException. There is a catch block for GreenException in main, so that catch block is executed (printing “main-green,”) then normal execution continues just below that catch blocks in main. The finally block comes next, so “main-finally,” is printed. Then]main is printed. Notice that after the exception is handled, we continue execution in main, not where the exception happened!

3. methodD throws a GreenException?

main[
A[
B[after C,B-finally,main-green,main-finally,]main

methodC doesn’t throw an exception, so “after C,” is printed. Aftering entering the try block in methodB, methodD is called. A GreenException is thrown, so normal execution terminates. The catch blocks in methodB don’t catch a GreenException, so java moves on to methodA. Since code in finally blocks is always run, the finally in methodB is run before moving to methodA. So “B-finally,” is printed.
The methodB is called in the try block in methodA, so the catch block in methodA can be considered. It doesn’t catch a GreenException, so main is considered next. The catch block for GreenException in main catches the exception and prints “main-green,”. The finally block in main prints “main-finally,” and normal execution continues after the catch blocks.]main is printed.

4. methodD throws a RedException?

main[
A[
B[after C,B-red,B-finally,]B
after B,]A
after A,after E,main-finally,]main

Just like in problem 3, an exception isn't thrown until methodD is called in the try block in methodB. Everything is the same up to that points, so I'll start explaining this problem from there.
Java looks at the catch blocks in methodB. There is one that catches RedException and it prints "B-red,". Then normal execution continues after the finally block prints "B-finally," because the exception has been handled.]B and a newline are printed.
Back in the try block of methodA, "after B," is printed. The catch blocks are skipped, and]A and a newline are printed.
In the try block in main, "after A," is printed. methodE doesn't throw an exception, so "after E," is printed. Then the finally in main prints "main-finally".]main is printed.

5. methodD throws a YellowException?

main[
A[
B[after C,B-yellow,B-finally,main-green,main-finally,]main

This problem is the same as the problems 3 and 4 until methodD is called. Starting from there, methodD throws a YellowException. That exception is caught in the 1st catch block in methodB. The catch block prints "B-yellow," and throws a GreenException. Since another exception is thrown. Even though the GreenException is thrown in methodB, it is thrown in a catch block, not in the try block, so java will not look at the catch blocks in methodB to catch the GreenException. In other words, a catch block can only catch exceptions thrown in its matching try block. The finally block in methodB prints "B-finally," before moving on the the catch blocks in methodA. The catch block in methodA doesn't catch the exception, so it is caught in a catch block in main. The catch block prints "main-green," then the finally in main prints "main-finally," then]main prints.

6. methodD throws an OrangeException?

```
main[
A[
B[after C,B-finally,main-finally,Exception in thread "main" OrangeException
        at Test.methodD(Test.java:53)
        at Test.methodB(Test.java:37)
        at Test.methodA(Test.java:25)
        at Test.main(Test.java:7)
```

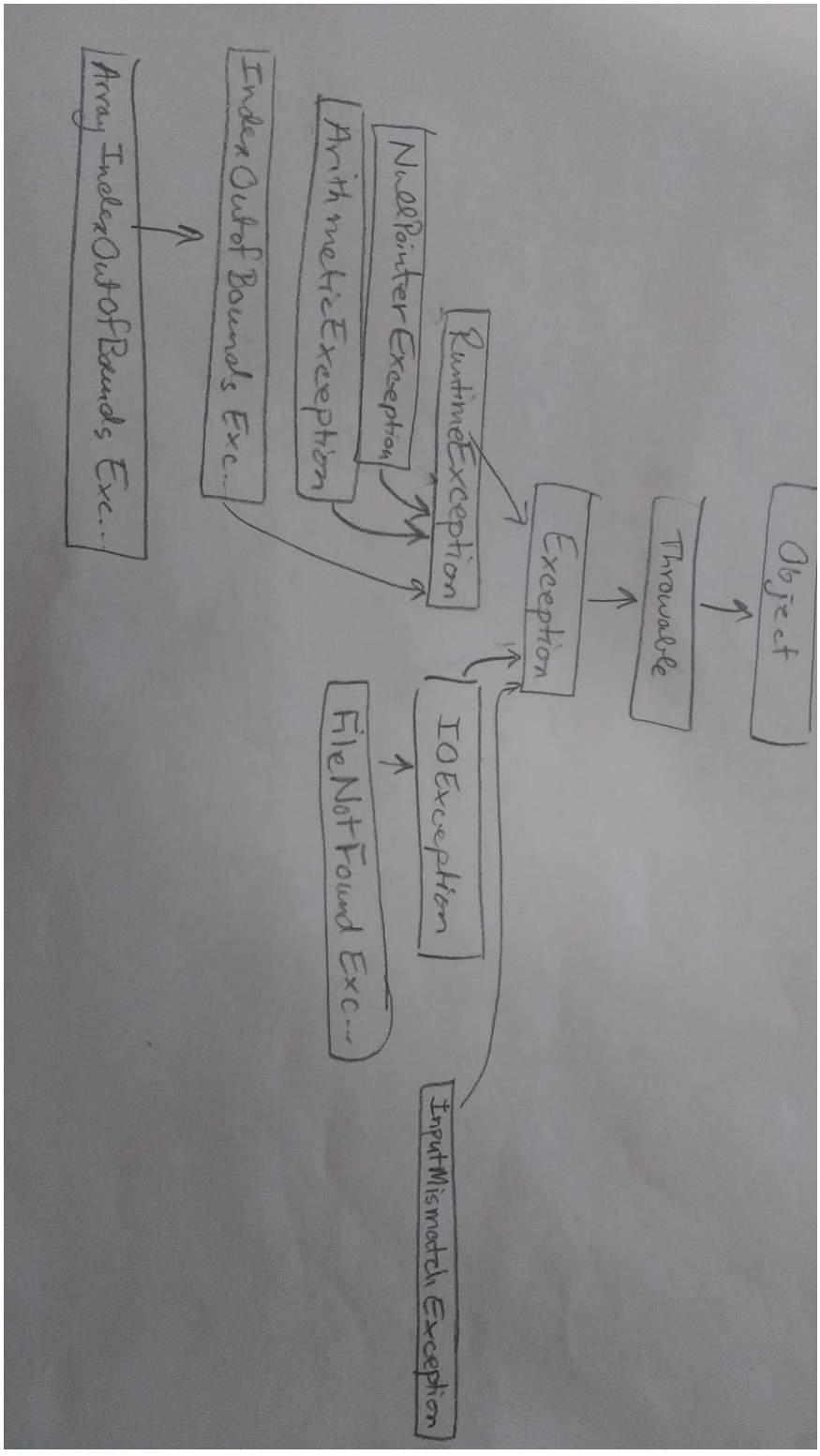
methodC doesn't throw an exception, so "after C," prints. methodD throws an exception which is not caught in the catch blocks in methodB, methodA, or main. After checking the catch blocks in methodB but before cheking the catch block in methodA, the finally block in methodB prints "B-finally,". Note that the finally block always executes before we leave the method, whether is an exception or not. In the normal case, it happens just after the try-block finishes, but in the error case, the finally executes right after the appropriate catch block executes OR before we exit the method to check if the parent catches the exception. After checking the catch blocks in main, the finally block in main prints "main-finally,".
Since the exception is not caught, the program will crash when run. The error message is printed.

7. A
Your code must handle all checked exceptions before the compiler will allow it to compile. This entails either catching the exception or adding a throws-clause (which declares the possibility of an error) to your method header, whichever is most appropriate. This forces the programmer to gracefully handle cases where things might go awry, making code more stable.

Checked exceptions extend Exception or one of its subclasses (except RuntimeException). Runtime (unchecked) exceptions extend the RuntimeException class or one of its subclasses. They do not need to be caught or declared for a program to compile, but if they aren't caught then at runtime the program will crash. That is what happened with the OrangeException in problem 6. OrangeException is an unchecked exception that wasn't caught, so the program crashed. This is also why you have never had to add "throws NullPointerException" to any of your programs; NullPointerException is a RuntimeException.

Note: Do not be deceived by names here. All exceptions happen at runtime! Errors that are reported at compile time are not exceptions; they are compile errors. The difference between checked exceptions and unchecked (runtime) exceptions does not have to do with when they occur. Rather, it has to do with how much the compiler will force you to do before it is willing to compile your program.

8.


Object
Throwable
Exception
RuntimeException
NullPointerException
ArithmeticException
IndexOutOfBoundsExc..
ArrayIndexOutOfBoundsExc..
IOException
FileNotFoundExc..
InputMismatchException

ArrayIndexOutOfBoundsException is an example of a runtime exception that doesn't directly extend RuntimeException but instead extends IndexOutOfBoundsException which extends RuntimeException. This is because ArrayIndexOutOfBoundsException is a special case of IndexOutOfBoundsException.

FileNotFoundException is an example of an unchecked exception that doesn't directly extend exception but instead extends IOException which extends Exception. This is because not finding a file deals with I/O.

All of the exceptions that extend RuntimeException are unchecked exceptions. Every other Exception in the inheritance tree is a checked exception.

Text File I/O:

1. ASCII characters (https://en.wikipedia.org/wiki/ASCII). ASCII is encoding of a set of standard characters into numbers. This includes letters, numbers, and some common symbols (e.g. !, #, ?, <, etc...). Each character has a unique 8-bit value which encodes it. For example 'A' = 65. Plain text files are always encoded in ASCII. Note that almost all programming languages use the ASCII encoding for your source code, which is why you can write a .java file in Eclipse and then open it in notepad and edit...

2. A file that contains other files (aka a folder).

3. The directory that the program was run in. For Eclipse, this is usually the project directory (usually called workspace). The current directory is important because it affects where your program thinks other files are relative to itself.

4. . (one dot) is shorthand for the current directory, and .. (two dots) is the shorthand for the parent directory (the one that contains the current directory (aka ".")).

5. The directory that has no parent. For example, c:\\ on Windows or / on macOS and other Unix-based systems. Most popular file systems have tree-shaped directory hierarchies -- that is, there is a unique root, and directories cannot contain themselves.

6. The sequence of directories to get from some directory to another file. Usually, we write these as a sequence of directory names separated by forward slashes (Unix/maxOS/Linux) or backslashes (Windows). For example: "foo/bar/baz" or "foo\bar\baz".

7. If the path is absolute, you are going from root to the current directory or to a file. If the path is relative you are going from the current directory to another directory or file. Absolute paths always start from the root. For example, "C:\\foo\bar\baz" or "/foo/bar/baz". Here, "C:\" and the leading "/" denote the root directory. Relative paths start from the current directory. If a path does not include the "C:\\" or leading "/", it is assumed to be relative.

8. Arguments passed to a program's main method. Most commonly this is done by invoking the program through the command line, but it can be done in Eclipse by using the "Run Configuration" panel.

9. Whenever your program uses a system resource you should release it when you are done. This includes files or System.in. For example, to close a Scanner scnr, you would use `scnr.close()`. This prevents the program from wasting system resources.scnr.close();

10. a. write.close() at the end of the try block

As noted in question 9, programs should release system resources that they use. PrintWriters use file system resources so PrintWriters should be closed. It is important for write.close(); to go in the try block. write is initialized in the try block,so before the try block, write is uninitialized. If an exception is thrown in the try block before initializing write, then write will still be uninitialized after the try catch blocks finish executing. Thus, the compiler will complain about uninitialized variables. This makes sense because if there was an error trying to obtain a resource, then there is no reason why we should close it!

b. FileNotFoundException ex

FileNotFoundException is a checked exception, so if it is possible for it to be thrown, there must be a catch block for it or the method must have throws clause. It is possible for a FileNotFoundException to be thrown in the line where a new file is made. That is because the java compiler cannot guarantee that input.txt exists until the program is actually run. Therefore, the compiler will force you to handle or declare the exception.

c. It exists, and its contents are “Hello World!Hello World!Hello World!”

Since input.txt exists, opening it and creating a scanner that reads from that file works (no exception is thrown). The PrintWriter is set to write to a file called output.txt. If output.txt doesn’t exist, the PrintWriter will create it. The while loop runs as long as there are more lines to be read from input.txt, and it writes each line of input.txt to output.txt. Since calling scnr.nextLine() reads in the entire line except the newline character, output.txt will have the same contents as input.txt but all on one line.

Interfaces and Abstracts:

1. It is not abstract and does not define method area.
   Some (or all) of the methods in an abstract class are abstract methods. Abstract methods are defined in the abstract class but are not implemented in the abstract class. It is up to the class that extends the abstract class to implement the abstract methods. The only exception to this is if an abstract class, call it B, extends an abstract class, call it A. Abstract classes can extend other abstract classes and not implement the abstract methods. But then any non-abstract class extending B must implement the abstract methods from both A and B. This makes sense because abstract classes cannot be directly instantiated (only their non-abstract children can be), so we are guaranteed that the compiler will always be able to find an implementation of abstract methods to call for an object.
   Since shape is abstract and has an abstract method called area, for Circle (which isn't an abstract class) to extend Shape, Circle must implement the area method.

2. We would want Shape to be abstract because it does not make sense for anyone to create a Shape object.
   All shapes have similar properties (such as they all have colors and they all can be compared to each other by comparing their areas), so it makes sense to have a way for all shapes to share their common properties. Making Shape a class is a great way to do this.
   On the other hand, different shapes have unique properties (such as how their area is calculated), so it doesn't make sense to view all shapes as the same. In order to allow shapes to share code for their common properties while keeping their individuality, we make Shape an abstract class. The commonalities between all shapes (such as them being compared based on their area) are implemented in Shape. The differences between shapes (such as how their area is calculated) are made abstract methods, so each different shape has to fill in the details. This improves code reuse and encodes some information about the structure of your program in the type system itself. That is, the Java compiler will be aware that there are a bunch of objects of subclasses of Shape that share some properties. The compiler can then check that they do indeed have these properties, preventing all sorts of weird errors!

3. None. Because Shape implements Comparable, all of its subclasses are Comparable.
   Shape's compareTo method compares the area of shapes. Even though all shapes calculate area's differently, all shapes have an area. Therefore, all shape's areas can be compared with each other's. This is part of the beauty of inheritance! You can extend Shape by providing an area method and voila! All of a sudden you are comparable with ANY Shape for free.

4. No. The constructor hard-codes the color, and Square overrides the setColor method to disallow changing the color.
   Although Shape allows colors to be set and Square extends Rectangle which extends Shape, Square prevents both ways of setting the color (the constructor and the setColor method). The Square constructor always uses the color blue. The only way to change the color is with the set method, but Square overrides the setColor method in such a way that attempting to change the color will results in an exception being thrown.

5. We can then reuse the code of the constructor to do anything that all shapes need to do during construction. For example, here it is setting the color.
This way we do not need to copy the code to set the color into the constructors for all the shapes that extend Shape. Moreover, we don't need to give any subclasses of Shape access to the String color field, reducing that chance that we do something silly and create a bug.

6. Same as #5, code reuse and modularity -- the key points of OOP.
By reuse, I mean that we do not need to have the code that sets color or compares shapes by their area in all the subclasses of Shape. We only need that code once in the Shape class.
By modularity, I mean different parts of the code can handle different functionalities. Shape has all the code needed to compare shapes or to set a shape's color.