HOMEWORK PROBLEMS 12

1) If an exception is not caught, what happens?

2) Write a catch block that catches `RuntimeException`. The catch block should display the message embedded in the exception it catches.

3) Write a statement that creates and throws a `RuntimeException` whose error message is "`Don't worry.`"

4) What is the difference between a checked and an unchecked exception?

5) Write an exception class named `InvalidNumber`. Its constructor should call its superclass, passing it the message "`Invalid Number`".

6) What is the effect of the following statement:

```
System.exit(1);
```

   Include it in a program and see what happens when it is executed. By convention, any non-zero argument used in a call of `exit` indicates an abnormal termination. A `0` argument indicates a normal termination.

7) Add `throws ArithmeticException` clauses to the methods in `C12h7.java` (a copy of `TestException1` in Fig. 12.1). Do these unnecessary clauses cause compile-time errors? Now replace the `throws ArithmeticException` clauses with `throws IOException` clauses. Also import `java.io.IOException`. Do these incorrect `throws` clauses cause compile-time errors?

8) Write a program that includes the method below and an `import` statement for `java.io.IOException`.

```
public static void f()
{
   throws new IOException("hello");
}
```

   What problem occurs during compile time? Fix the problem.

9) Repeat homework problem 8, but use `RuntimeException` in place of `IOException`. You do not need to import `RuntimeException`. What happens at compile time? Why is the result different from the result homework problem 8?

10) Replace the `f` method in `C12h10.java` (a copy of `TestException1` in Fig. 12.1) with

```
public static void f()
{
   try
   {
      System.out.println("Start of f");
      g();
      System.out.println("End of f");
   }
   catch (ArithmeticException e)
   {
      System.out.println("In catch block");
      System.out.println(e.getMessage());
   }
```

```
      finally
      {
         System.out.println("In finally block");
      }
   }
```

Determine what happens for each of the three cases below. For which of these cases is the `finally` block executed?

1) No exception is thrown in the `try` block.

2) An exception is thrown and caught by the `catch` block.

3) The exception is thrown but not caught (which happens if the thrown exception does not match the type of exception specified in the catch block). To try out this case, change the type of the parameter `e` in the code above to `IOException`. Import `java.io.IOException`.

11) Change the statement on line 28 in `C12h11.java` (copy of `TestException2` in Fig. 12.2) to

```
System.out.println(e);
```

Is the output displayed by the program different? What method provides the string that is displayed?

12) When you create a new `RuntimeException`, can you call its constructor without using any arguments? Run a test program to check your answer.

13) What kind of exception occurs when the following program is executed?

```
class C12h13
{
   public static void main(String[] args)
   {
     NullPointer r;
     r = null;
     r.f();
   }
}
//================================================
class NullPointer
{
    public void f()
    {
       System.out.println("hello");
    }
}
```

14) Write a program in which `main` calls a method that uses a too large index when accessing an array. Let the exception propagate until it terminates the program. What kind of exception occurs? Now include a `catch` block for this type of exception in `main`. Your `catch` block should display a meaningful error message in addition to the message embedded in the exception. It should then terminate the program immediately (see homework problem 6).

15) `RuntimeException` is a superclass of both `ArithmeticException` and `NullPointerException`. Will a `catch` block whose parameter has type `RuntimeException` catch both `ArithmeticException` and `NullPointerException` exceptions? Run a test program to check your answer.

16) Given the code below, what would happen if an `ArithmeticException` occurred in the `try` block? A `NullPointerException`? A `RuntimeException`? Hint: `ArithmeticException`, `NullPointerException`, and `RuntimeException` are all subclasses of `Exception`. Would it make sense to reverse the order of the two `catch` blocks? Run a test program to check your answers.

```
try
{
   ...
}
catch(Exception e)
{
   ...
}
catch(NullPointerException e)
{
   ...
}
```

17) Write a program that prompts for and reads in a string. It should throw a `NoEException` (a class you define) if the string has no occurrences of the letter E (either upper or lower case). Test your program.

18) Write a program in which `main` calls a non-static method `f`. `f` should prompt the user for and read in an integer from the keyboard. If the integer is negative, `f` should construct and throw an `IOException` (you can do this even though an I/O error has not actually occurred). If the integer is positive, `f` should construct and throw a `NullPointerException`. If the integer is zero, `f` should construct and throw an `ArithmeticException`. `main` should catch the three types of exceptions with separate `catch` blocks. It should then display one of the following messages according to the type of exception it catches, after which the program should terminate:

```
IOException caught
NullPointerException caught
ArithmeticException caught
```

19) Write a program in which `main` prompts for and reads in a `double` constant and then displays the square of the constant. If the user does not enter a `double` constant, your program should re-prompt the user for a valid input. Your program should use the prompt and error messages illustrated by the following sample session:

```
Enter non-negative double constant
hello
Not a double constant. Re-enter
goodbye
Not a double constant. Re-enter
4
Square of 4 = 16.0 ◄——— program terminates at this point
```

Do not use `hasNextDouble` to determine if the input is valid. Instead, catch the exception that `nextDouble` throws if the input is invalid.

20) Write a program that adds and displays the sum of the command line arguments. If no command line arguments are given, your program should display

```
Error: no command line arguments
```

and then terminate execution. If any line argument is not a valid number, your program should display

```
Error: invalid command line argument
```

and then terminate execution.

21) Write a class named `IntCompute`. It should contain an `add`, a `subtract`, a `multiply`, a `divide`, and a `remainder` method. Each of the methods should perform the operation implied by its name on its two parameters and return the result. However, if the result is out of the range of type `int`, then it should throw an `Overflow` exception. Define the `Overflow` class appropriately. Write a program that uses your `IntCompute` class. Your program should call all the methods in `IntCompute,` both with arguments that do not cause overflow and with arguments that do.

22) Create a class named `CandyBars` that throws a `TooManyCandyBars` exception if the creation of more than the allowed limit of `CandyBars` objects is attempted. `CandyBars` should have two constructors: one with no parameters that sets the limit at 30, and one with one parameter that sets the limit to the value of this parameter. Write a program that tests your `CandyBars` class.

23) Write a class named `Print`. It should contain a a `print` and a `println` method that work like `System.out.print` and `System.out.println`. However, the methods in `Print` should throw a `TooManyLines` exception if the total number of lines displayed exceeds 25. Define the `TooManyLines` class appropriately. Write a program that tests your `Print` class.