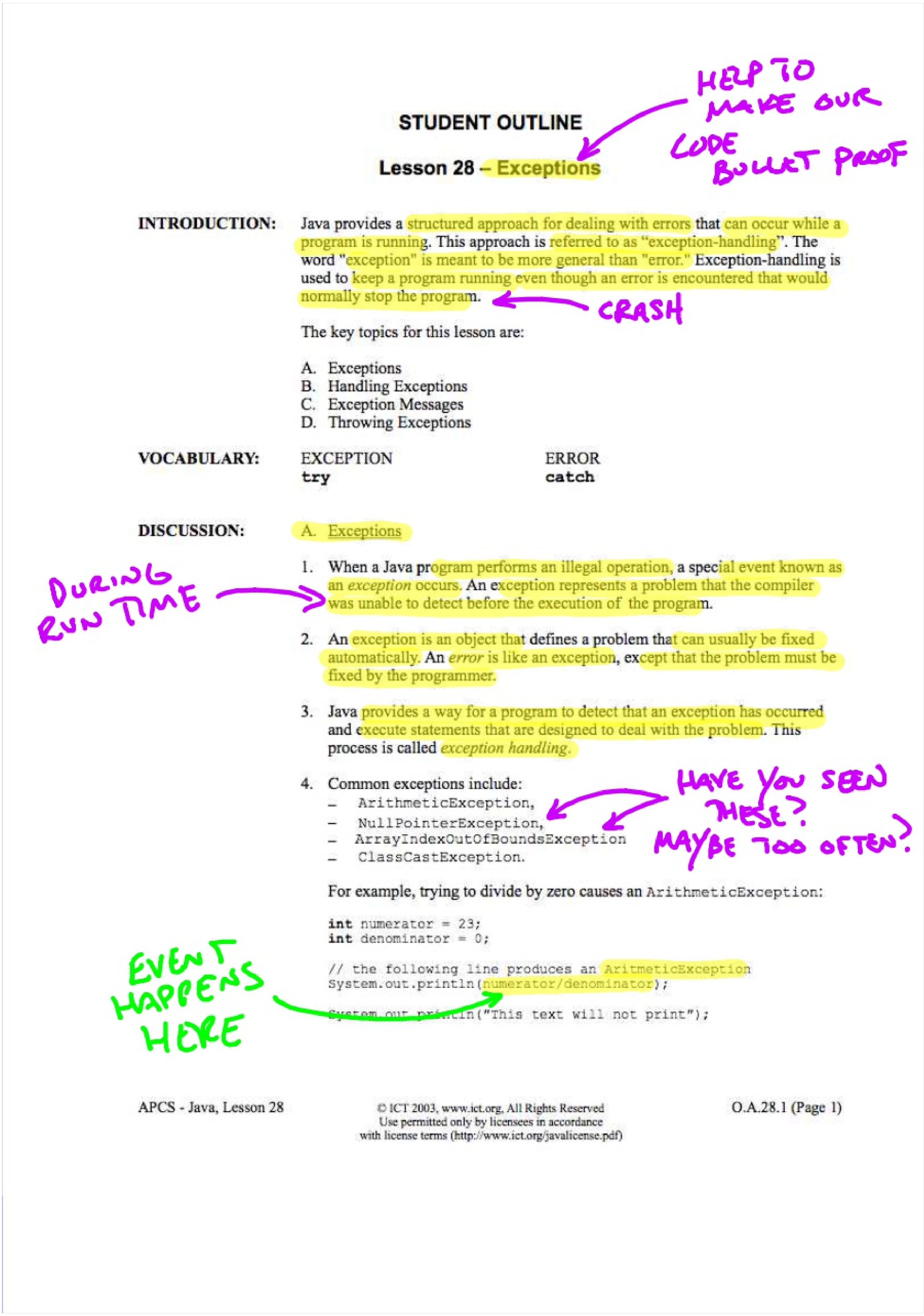
What is the output from the following code segment??

```
Comparable x = new Integer(125);
System.out.print(x.compareTo("123"))
;
```

I) 0
II) A positive integer
III) A syntax error on line 1
IV) A syntax error on line 2
[✓ V) A CastClassException]



STUDENT OUTLINE

Lesson 28 – Exceptions

HELP TO MAKE OUR CODE BULLET PROOF

INTRODUCTION: Java provides a structured approach for dealing with errors that can occur while a program is running. This approach is referred to as "exception-handling". The word "exception" is meant to be more general than "error." Exception-handling is used to keep a program running even though an error is encountered that would normally stop the program.

CRASH

The key topics for this lesson are:

A. Exceptions
B. Handling Exceptions
C. Exception Messages
D. Throwing Exceptions

VOCABULARY:	EXCEPTION	ERROR
	try	**catch**

DISCUSSION: A. Exceptions

DURING RUN TIME

1. When a Java program performs an illegal operation, a special event known as an *exception* occurs. An exception represents a problem that the compiler was unable to detect before the execution of the program.

2. An exception is an object that defines a problem that can usually be fixed automatically. An *error* is like an exception, except that the problem must be fixed by the programmer.

3. Java provides a way for a program to detect that an exception has occurred and execute statements that are designed to deal with the problem. This process is called *exception handling*.

4. Common exceptions include:
   - `ArithmeticException`,
   - `NullPointerException`,
   - `ArrayIndexOutOfBoundsException`
   - `ClassCastException`.

HAVE YOU SEEN THESE? MAYBE TOO OFTEN?

For example, trying to divide by zero causes an `ArithmeticException`:

```
int numerator = 23;
int denominator = 0;

// the following line produces an AritmeticException
System.out.println(numerator/denominator);

System.out.println("This text will not print");
```

EVENT HAPPENS HERE

A NullPointerException occurs if you use a null reference where you need an object reference. For example,

```
String name = null;

// the following line produces an NullPointerException
if(name.length() > 0)
  System.out.println("This text will not print");
```

THIS CRASHES BECAUSE OF THIS

Since name has been declared to be a reference to a String and has the value null, indicating that it is not referring to any String at this time, an attempt to call a method within name, such as name.length(), will cause a NullPointerException.

We will discuss the remaining two exceptions, ArrayIndexOutOfBoundsException and ClassCastException, in later lessons.

ON FRONT PAGE!!

B. Handling Exceptions

1. When a exception occurs (we say that the exception is *thrown*), the program has the option of *catching* it. To *catch* an exception, we must anticipate where the exception might occur and enclose that code in a **try** block. The try block, is followed by a **catch** block that catches the exception (if it occurs) and performs the desired action.

2. The general form of a **try-catch** statement is:

```
try
{
  try-block
}
catch (exception-type identifier)
{
  catch-block
}
```

ATTEMPT AN ACTION

THIS IS AN ARGUMENT

EXCEPTION IS FOUND

HOW WE DEAL WITH IT.

   a. The *try-block* refers to a statement or series of statements that might throw an exception.
   b. The *catch-block* refers to a statement or series of statements to be executed if the exception is thrown. A **try** block can be followed by more than one **catch** block. When an exception is thrown inside a **try** block, the first matching **catch** block will handle the exception.
   c. *exception-type* specifies what kind of exception object the **catch** block should handle.
   d. *identifier* is an arbitrary variable name used to refer to the *exception-type* object.

3. The **try** and **catch** blocks together form a single statement, which can be used anywhere in a program that a single statement is allowed.

④ KEEP YOUR TRY BLOCK NARROW & LOOK FOR SPECIFIC PROBLEMS

IMMEDIATE TERMINATION OF THAT BLOCK!

4. If an exception is thrown anywhere in the **try** block, and the exception matches the one named in the **catch** block, then the code in the catch block is executed. If the **try** block executes normally, without an exception, the **catch** block is ignored.

5. Here is an example of **try** and **catch**:

```
import chn.util.*;

public class CheckDivideByZero
{
  public static void main (String[] args)
  {
    ConsoleIO console = new ConsoleIO();
    int quotient;

    System.out.print("Enter the numerator: ");
    int numerator = console.readInt();

    System.out.print("Enter the denominator: ");
    int denominator = console.readInt();
    try
    {
      quotient = numerator/denominator;
      System.out.println("The answer is: " + quotient);
    }
    catch (ArithmeticException e)
    {
      System.out.println("Error: Division by zero");
    }
  }
}
```

POTENTIAL PROBLEM EVENT

NO "CRASH" OPPORTUNITY TO RECOVER

Run Output:

```
Enter the numerator: 23
Enter the denominator: 0
Error: Division by zero
```

If the value of `denominator` is zero, then an `ArithmeticException` will be thrown when `numerator` is divided by `denominator`. The **catch** block will catch the exception and print an error message. If the value of `denominator` is not zero, the code in the **catch** block will be ignored. Either way, the program continues executing at the next statement after the **catch** block.

C. Exception Messages

1. If a program does not handle exceptions at all, it will crash and produce a message that describes the exception and where it happened. This information can be used to help track down the cause of a problem.

2. The program shown below throws an `ArithmeticException` when the program tries to divide by zero. The program crashes and prints out information about the exception:

```
public class DivideByZero
{
  public static void main (String[] args)
  {
    int numerator = 23;
    int denominator = 0;

    // the following line produces an AritmeticException
    System.out.println(numerator/denominator);

    System.out.println("This text will not print");
  }
}
```

RUN CRASHES YIELDS

STACK TRACE

EXCEPTION MESSAGE

Run Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZero.main(DivideByZero.java:10)
```

The first line of the output tells which exception was thrown and gives some information about why it was thrown.

The rest of the output tells where the exception occurred and is referred to as a *call stack trace*. In this case, there is only one line in the call stack trace, but there could be several, depending on where the exception originated. The trace line gives the method, file, and line number where the exception happened.

3. When exceptions are handled by a program, it is possible to obtain information about an exception by referring to the "exception object" that Java creates in response to an exception condition. Every exception object contains a String that can be accessed using the `getMessage` method as follows:

WE CAN CALL THIS

```
try
{
  quotient = numerator/denominator;
  System.out.println("The answer is: " + quotient);
}
catch (ArithmeticException e)
{
  System.out.println(e.getMessage());
}
```

REMEMBER, THIS IS AN ARGUMENT

NO CRASH

If the exception is thrown, the following message is displayed

OUT TO CONSOLE

```
/ by zero
```

4. Printing the value returned by `getMessage` can be useful in situations where we are unsure of the type of error or its cause.

INTENTIONALLY

D. Throwing Exceptions

1. There are times when it makes sense for a program to deliberately throw an exception. This is the case when the program discovers some sort of exception or error condition, but there is no reasonable way to handle the error at the point where the problem is discovered. The program can throw an exception in the hope that some other part of the program will catch and handle the exception.

2. To throw an exception, use a **throw** statement. The syntax of the **throw** statement is:

```
throw exception-object;
```

3. For example, the following statement throws an `ArithmeticException`:

```
throw new ArithmeticException("Division by zero");
```

THE MESSAGE

The exception object is created with the **new** operator right in the **throw** statement. Exception classes in Java have a default constructor that takes no arguments and a constructor that takes a single `String` argument. If provided, this string appears in the exception message when the exception occurs.

4. In later lessons, we will look at exceptions thrown in the implementation of abstract data types.

SUMMARY/ REVIEW:

Exceptions provide a clean way to detect and handle unexpected situations. When a program detects an error, it throws an exception. When an exception is thrown, control is transferred to the appropriate exception handler. By defining a method that catches the exception, the programmer can write the code to handle the error.

ASSIGNMENT: Lab Exercise L.A.28.1, *ErrorCheck*.