Package and Exceptions

Session 8

Objectives

Packages

Exception Handling

- try block
- catch block
- finally block
- custom exception class

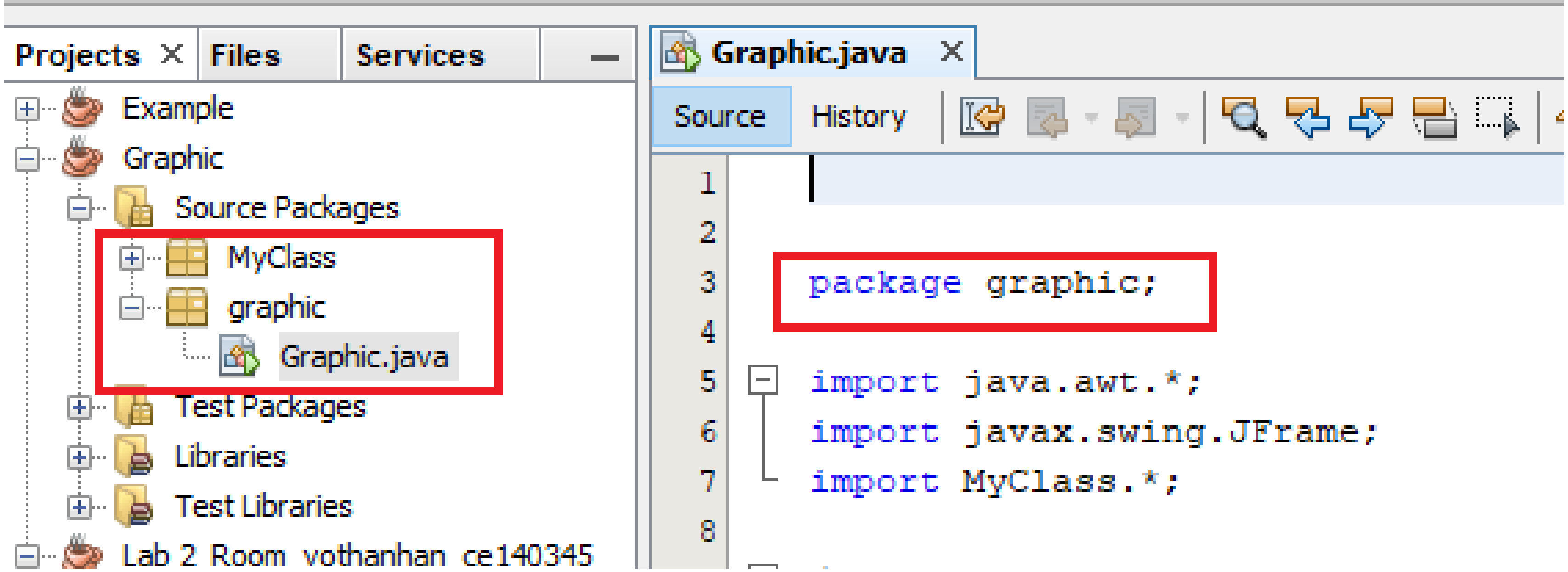
Assertions

Packages

- A **package** is a grouping of related classes, interfaces, enumerations, and annotation types providing access protection and name space management.
- Syntax to create a new package:

```
package [package name];
```

- This statement must be the first line in the source file.
- There can be only one package statement in each source file, and it applies to all types in the file.



```java

package graphic;

import java.awt.*;
import javax.swing.JFrame;
import MyClass.*;

```

Using Members of Packages

- Import the member's entire package

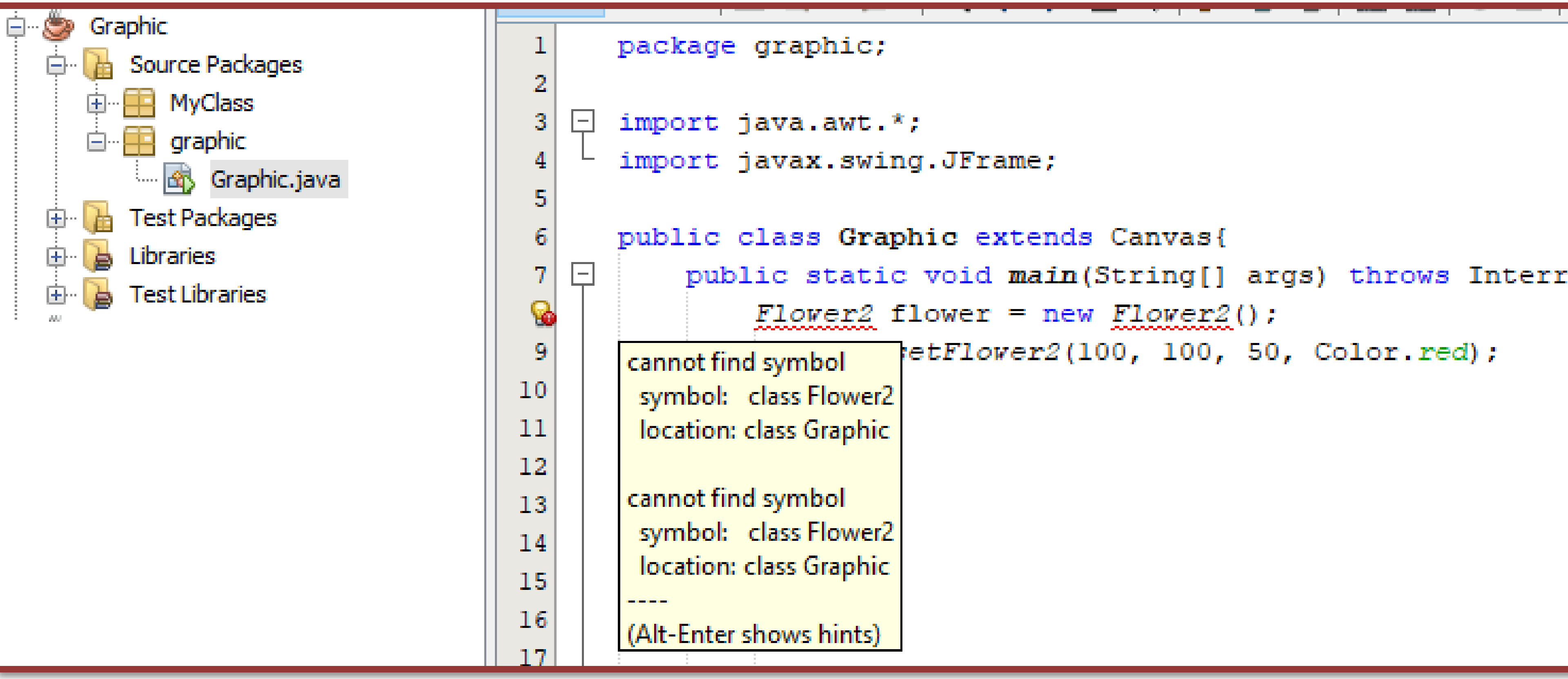
```
import graphics.*;

…

Rectangle myRectangle = new Rectangle();
```

- Import the package member

```
import graphics.Rectangle;

…

Rectangle myRectangle = new Rectangle();
```

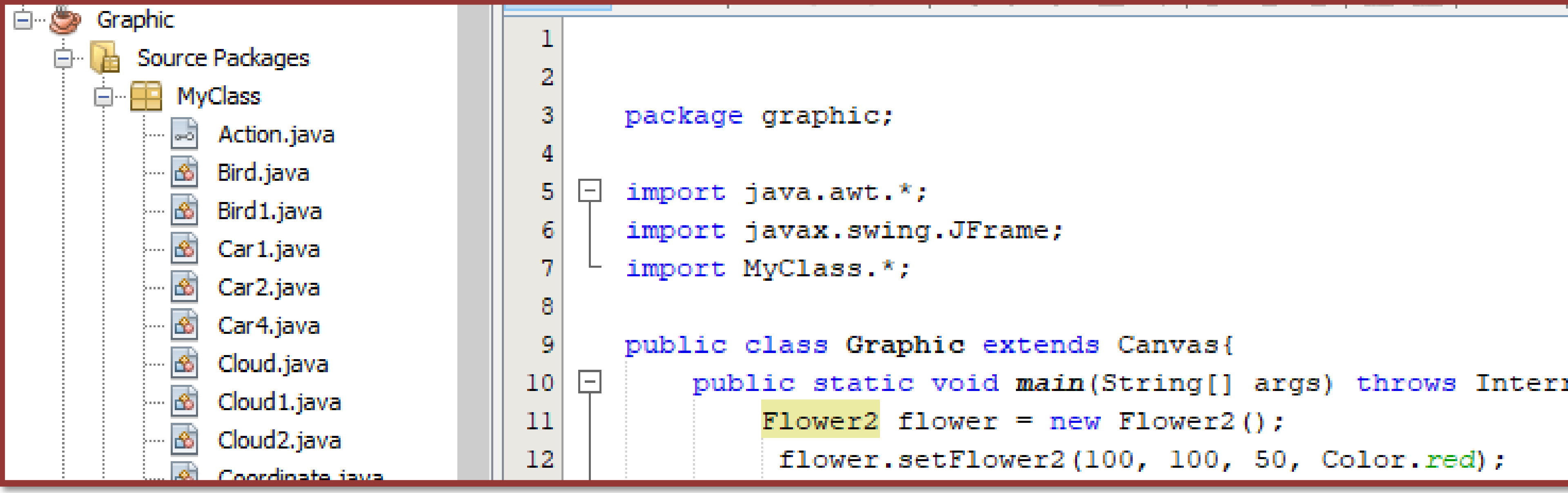
- Refer to the member by its fully qualified name

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

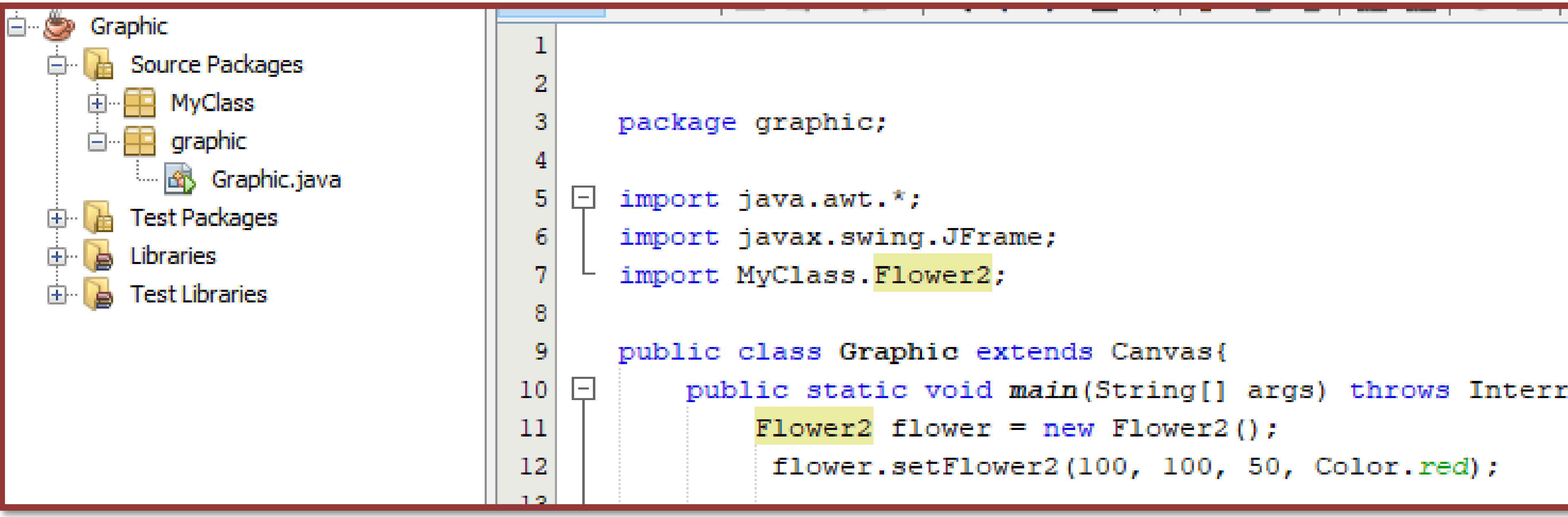


```
package graphic;

import java.awt.*;
import javax.swing.JFrame;

public class Graphic extends Canvas{
    public static void main(String[] args) throws Interr
        Flower2 flower = new Flower2();
                    etFlower2(100, 100, 50, Color.red);
```

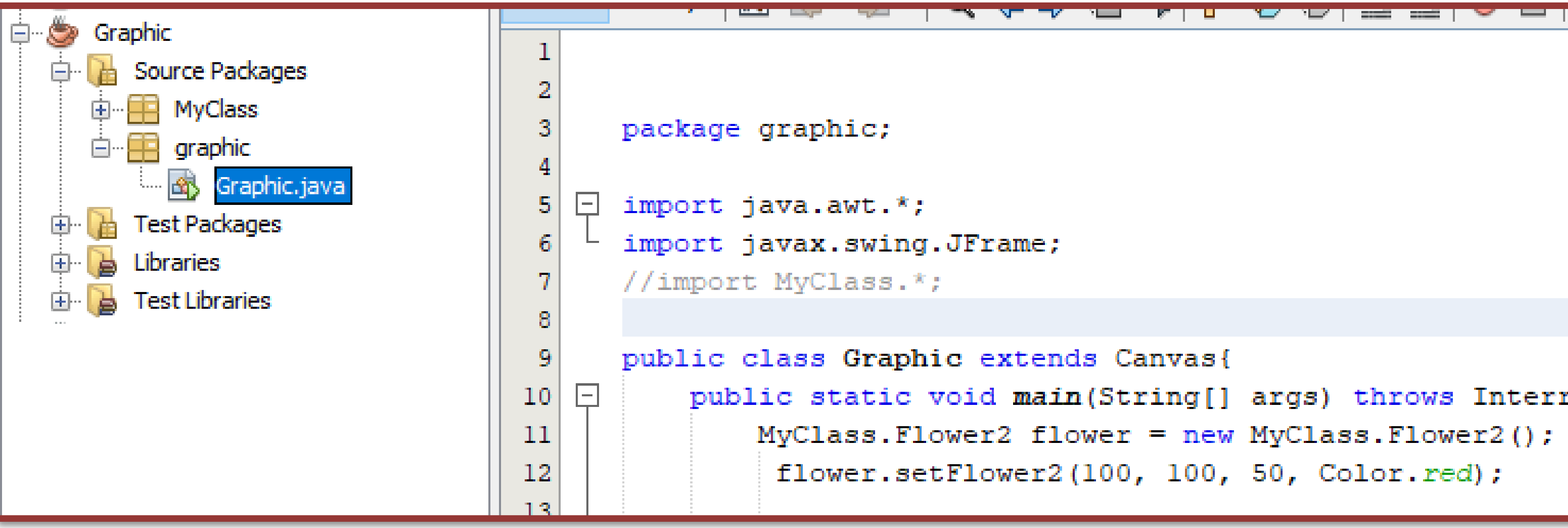
```
cannot find symbol
  symbol:   class Flower2
  location: class Graphic

cannot find symbol
  symbol:   class Flower2
  location: class Graphic
----
(Alt-Enter shows hints)
```



```java


package graphic;

import java.awt.*;
import javax.swing.JFrame;
import MyClass.*;

public class Graphic extends Canvas{
    public static void main(String[] args) throws Inter
        Flower2 flower = new Flower2();
         flower.setFlower2(100, 100, 50, Color.red);
```



```java


package graphic;

import java.awt.*;
import javax.swing.JFrame;
import MyClass.Flower2;

public class Graphic extends Canvas{
    public static void main(String[] args) throws Interr
        Flower2 flower = new Flower2();
         flower.setFlower2(100, 100, 50, Color.red);
```

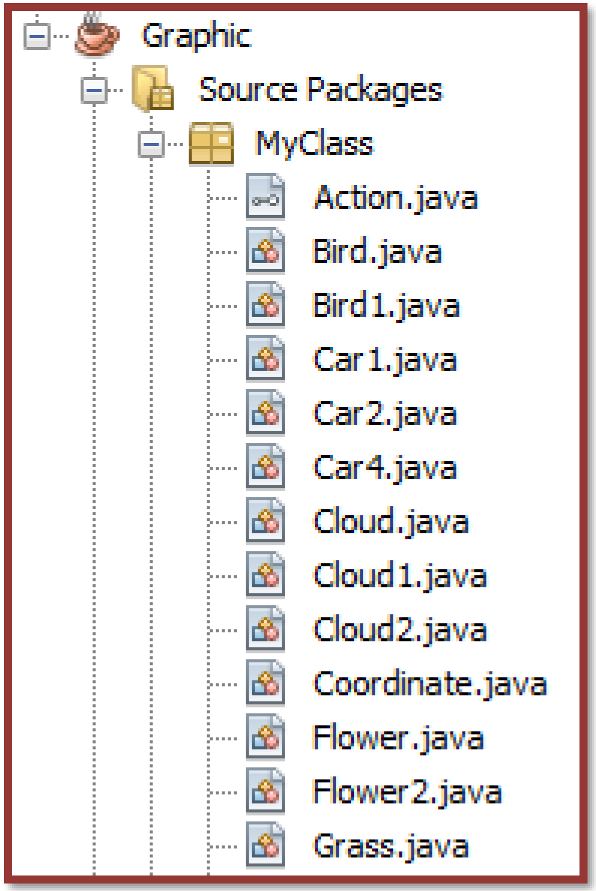


- Graphic
  - Source Packages
    - MyClass
    - graphic
      - Graphic.java
  - Test Packages
  - Libraries
  - Test Libraries

```

package graphic;

import java.awt.*;
import javax.swing.JFrame;
//import MyClass.*;

public class Graphic extends Canvas{
    public static void main(String[] args) throws Interr
        MyClass.Flower2 flower = new MyClass.Flower2();
         flower.setFlower2(100, 100, 50, Color.red);
```

???

- At the first line of each java file your code must be: ……

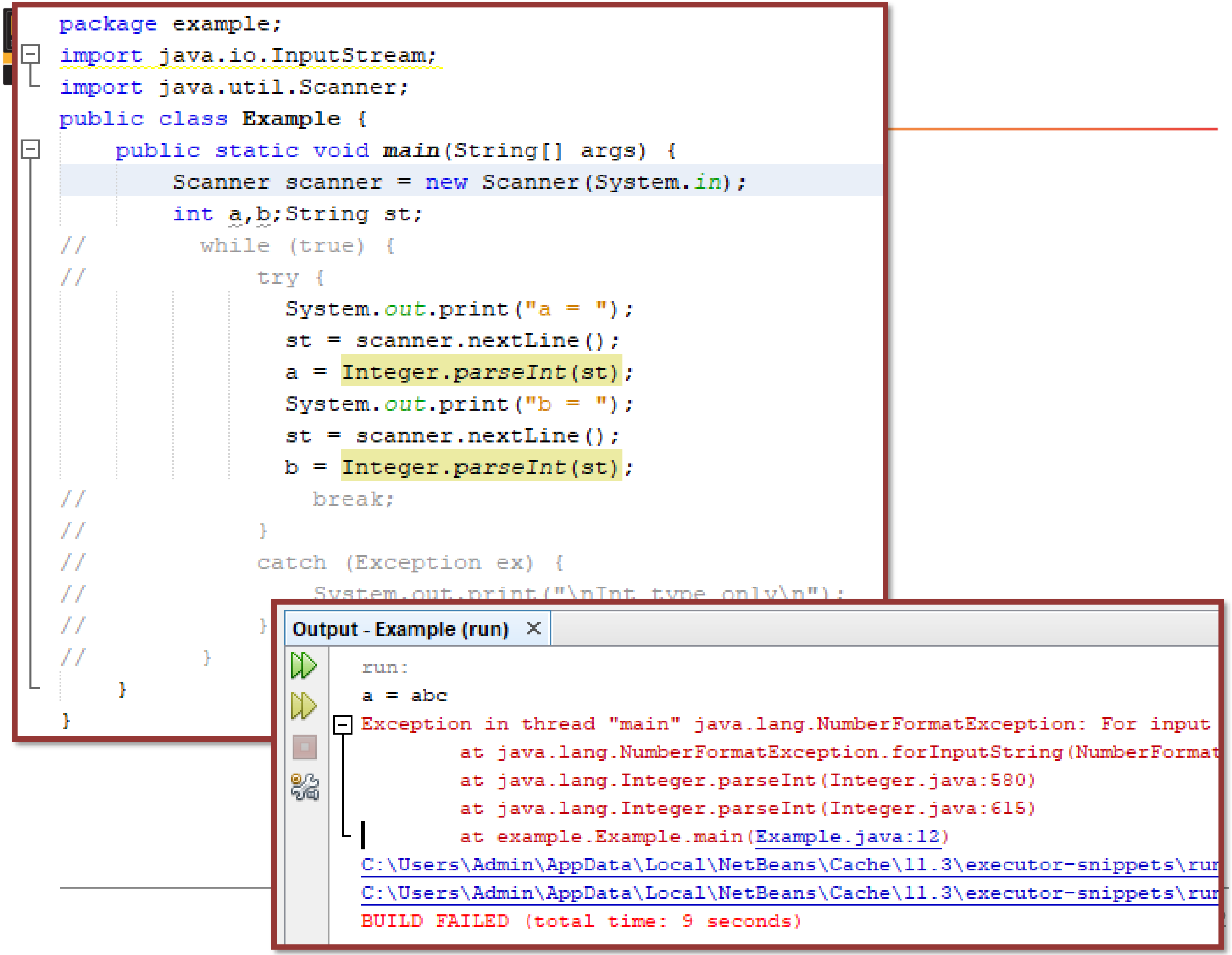


Packages

- Easily determine that these types are related.
- Know where to find types that can provide graphics-related functions.
- Won't conflict the names of your types with the type names in other packages because the package creates a new namespace.
- Allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

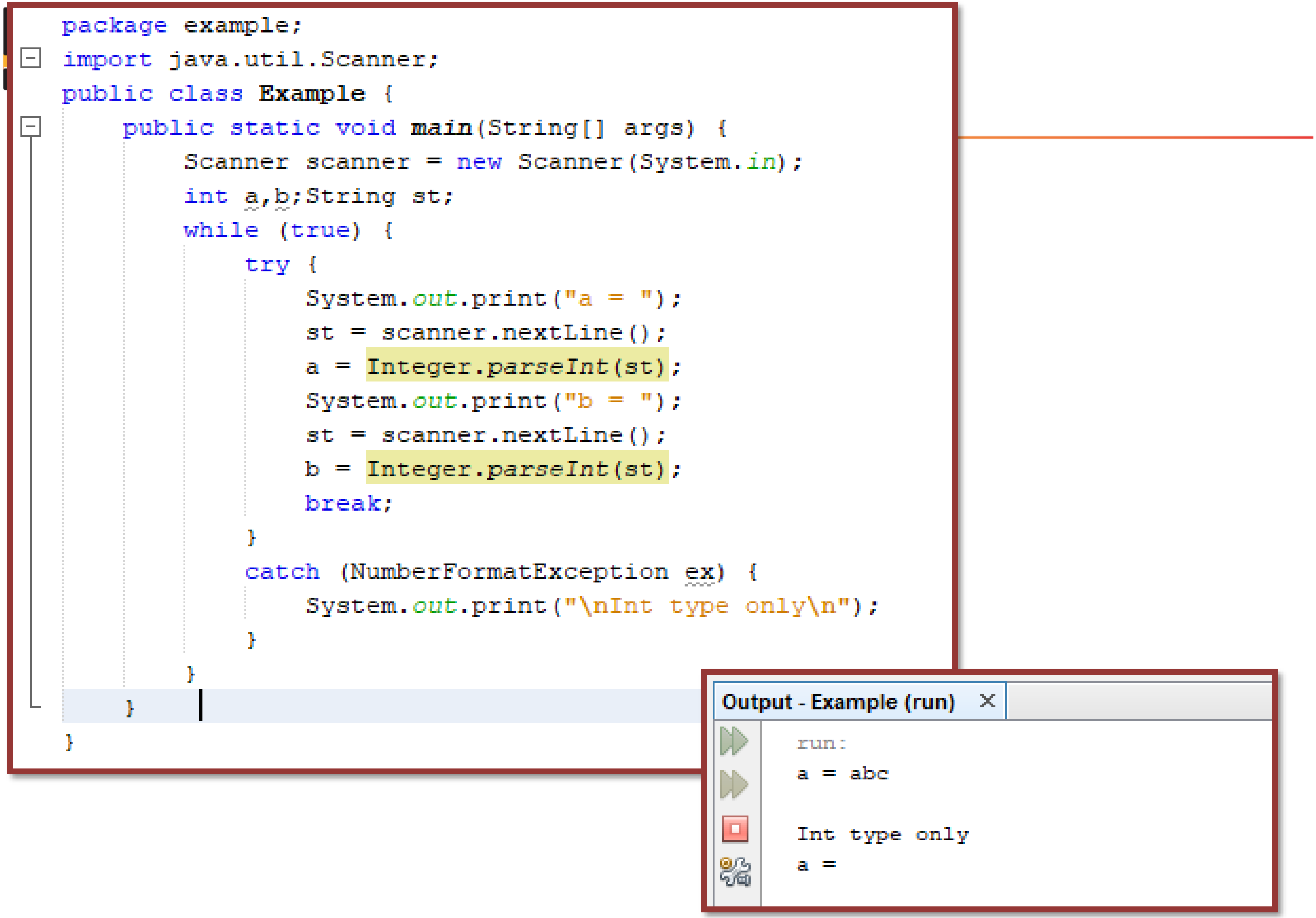
Exceptions

- Java is a very robust and efficient programming language (classes, objects, inheritance… -> a strong, versatile, and secure language).
- However, no matter how well a code is written, it is prone to failure or behaves erroneously in certain conditions.
- These situations may be expected or unexpected.
- In either case, the user would be nonplussed or confused with such unexpected behavior of code.
- To avoid such a situation, Java provides the concept of exception handling using which, a programmer can display appropriate message to the user in case such unexpected behavior of code occurs.



```
package example;
import java.io.InputStream;
import java.util.Scanner;
public class Example {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int a,b;String st;
//          while (true) {
//              try {
                    System.out.print("a = ");
                    st = scanner.nextLine();
                    a = Integer.parseInt(st);
                    System.out.print("b = ");
                    st = scanner.nextLine();
                    b = Integer.parseInt(st);
//                  break;
//              }
//              catch (Exception ex) {
//                  System.out.print("\nInt type only\n");
//              }
//          }
    }
}
```

```
run:
a = abc
Exception in thread "main" java.lang.NumberFormatException: For input
        at java.lang.NumberFormatException.forInputString(NumberFormat
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at example.Example.main(Example.java:12)
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\ru
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\ru
BUILD FAILED (total time: 9 seconds)
```



```java
package example;
import java.util.Scanner;
public class Example {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int a,b;String st;
        while (true) {
            try {
                System.out.print("a = ");
                st = scanner.nextLine();
                a = Integer.parseInt(st);
                System.out.print("b = ");
                st = scanner.nextLine();
                b = Integer.parseInt(st);
                break;
            }
            catch (NumberFormatException ex) {
                System.out.print("\nInt type only\n");
            }
        }
    }
}
```

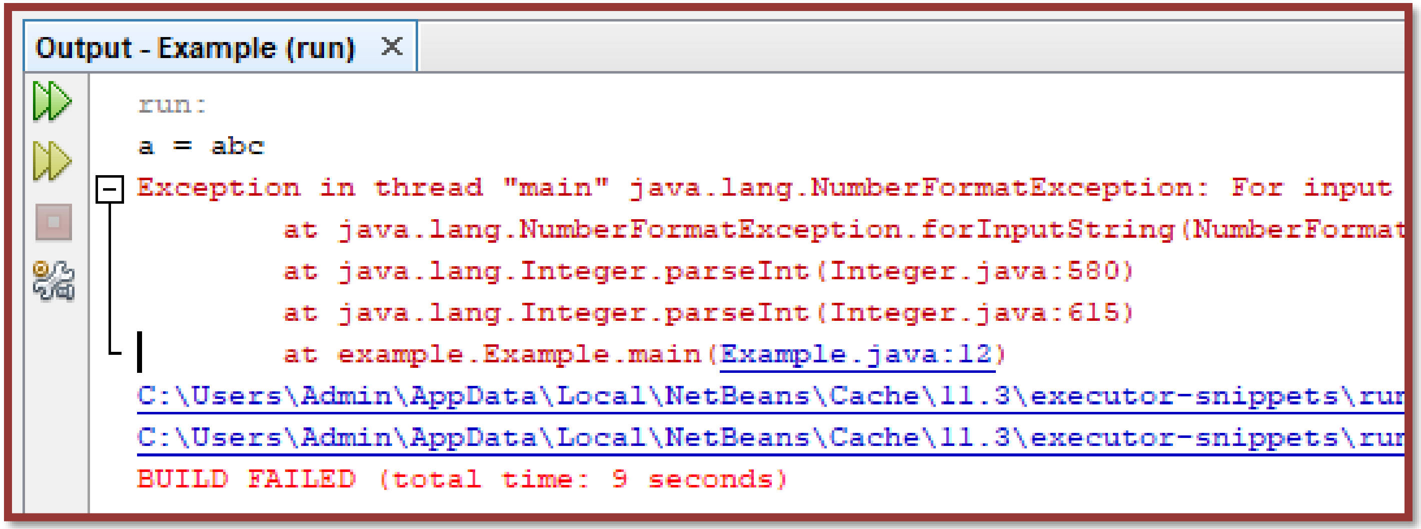
```
run:
a = abc

Int type only
a =
```

Exceptions

- An exception is an event or an abnormal condition in a program occurring during execution of a program that leads to disruption of normal flow of program instructions.
- An exception can occur for different reasons such as:
  - when the user enters invalid data
  - a file that needs to be opened cannot be found
  - a network connection has been lost in the middle of communications
  - the JVM has run out of memory

Exceptions

- When an error occurs inside a method, it creates an exception object and passes it to the runtime system.
- This object holds information about the type of error and state of the program when the error occurred.



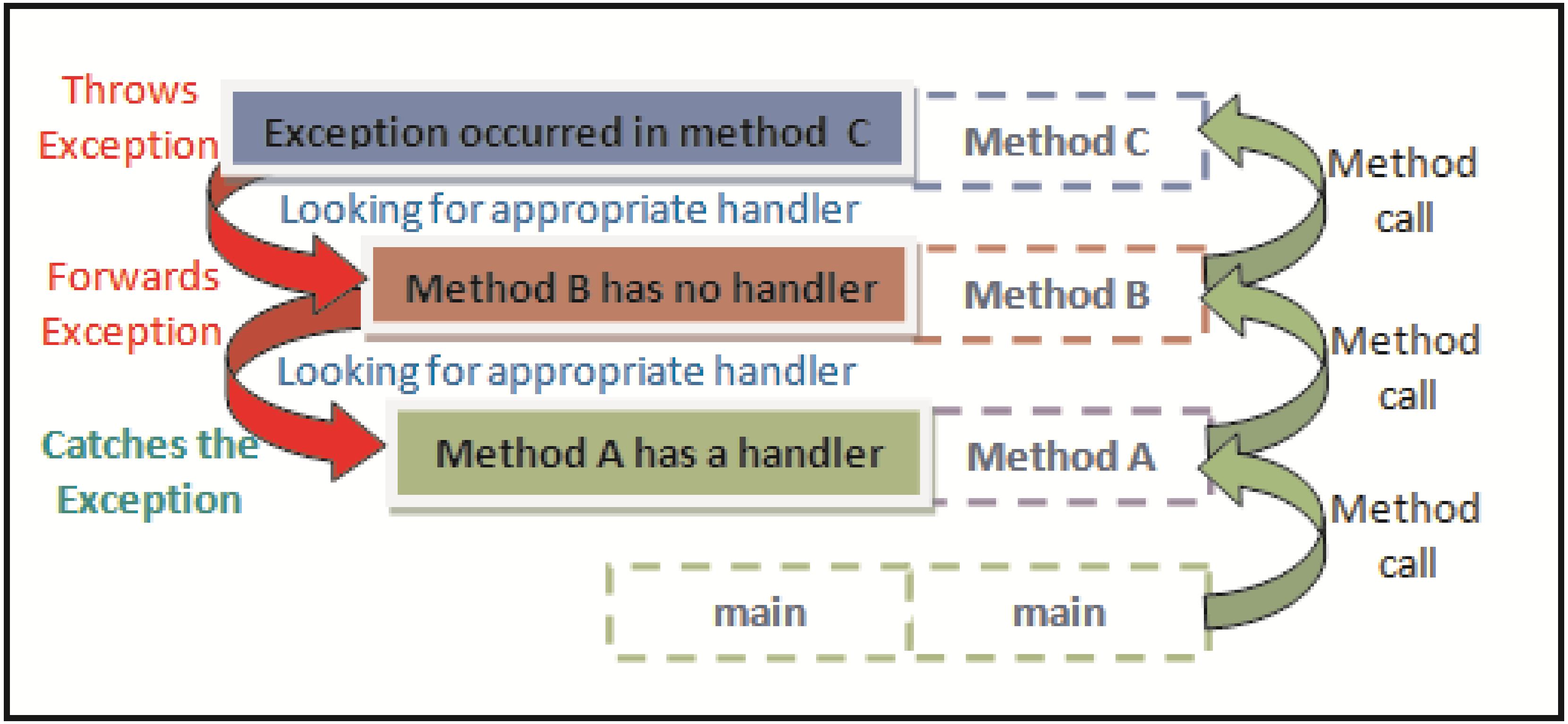
```
run:
a = abc
Exception in thread "main" java.lang.NumberFormatException: For input
        at java.lang.NumberFormatException.forInputString(NumberFormat
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at example.Example.main(Example.java:12)
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\run
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\run
BUILD FAILED (total time: 9 seconds)
```

Exceptions

- The process of creating an exception object and passing it to the runtime system is termed as throwing an exception.
- The runtime system tries to find some code block to handle it.
- The possible code blocks where the exception can be handled are a series of methods that were invoked in order to reach the method where the error has actually occurred.
- This list or series of methods is called the call stack. The stack trace shows the sequence of method invocations that led up to the exception.

Exceptions

- Following figure shows an example of method call stack:



The figure shows the method call from

main → Method A → Method B → Method C.

Types of Errors and Exceptions: Checked Exceptions

- These are exceptions that a well-written application must anticipate and provide methods to recover from.
- For example
  - suppose an application prompts the user to specify the name of a file to be opened and the user specifies the name of a nonexistent file.
  - In such a case, the `java.io.FileNotFoundException` is thrown.
- However, a well-written program will have the code block to catch this exception and inform the user of the mistake by displaying an appropriate message.
- In Java, all exceptions are checked exceptions, except those indicated by `Error`, `RuntimeException`, and their

Types of Errors and Exceptions: Unchecked Exception

- These are exceptions that are external to the application.
- The application usually cannot anticipate or recover from errors.
- For example:
  - suppose the user specified correct file name for the file to be opened and the file exists on the system.
  - However, the runtime fails to read the file due to some hardware or system malfunction.
  - Such a condition of unsuccessful read throws the `java.io.IOError` exception.
  - In this case, the application may catch this exception and display an appropriate message to the user or leave it to the program to print a stack trace and exit.

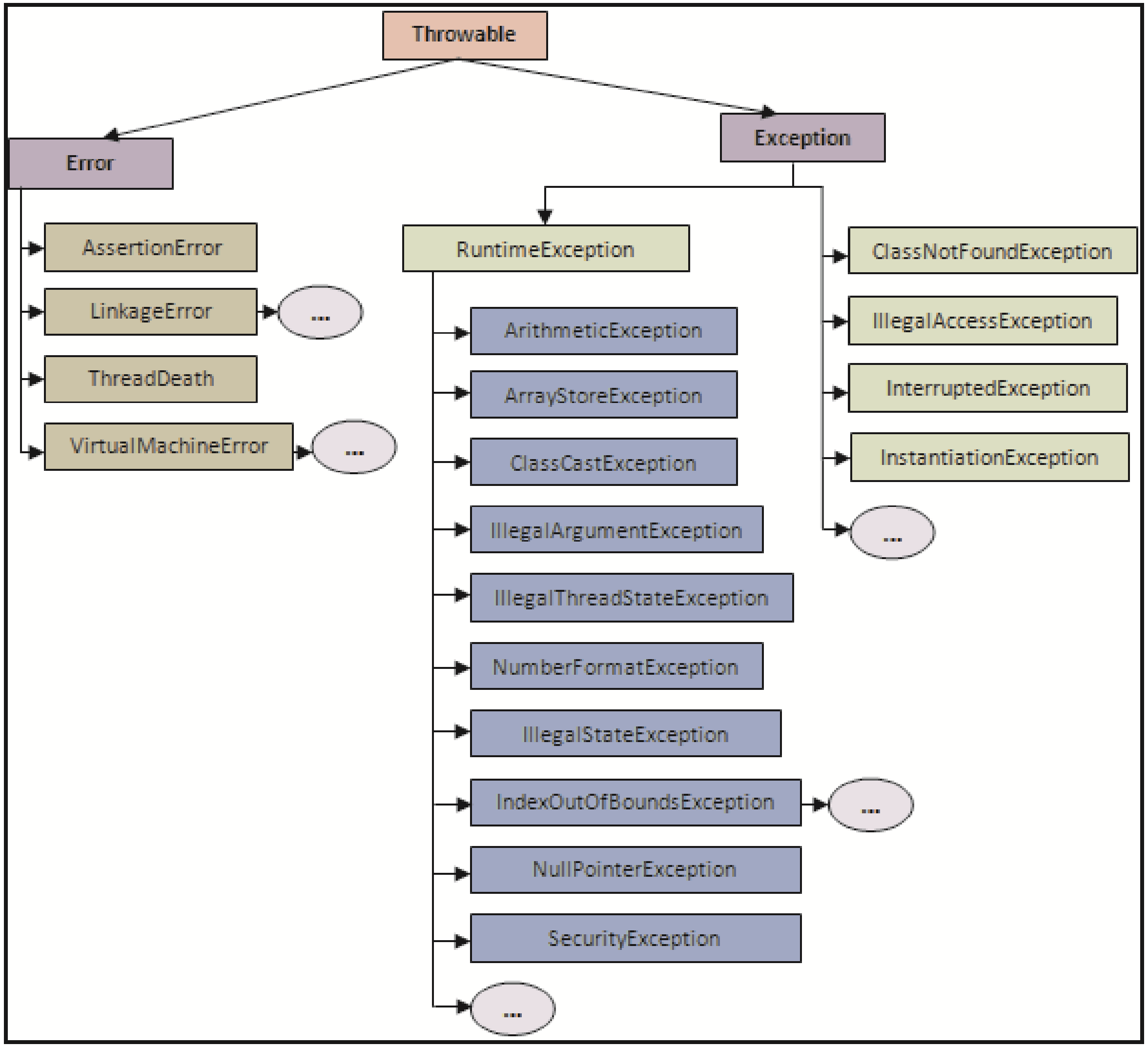
Types of Errors and Exceptions: Runtime Exception

- These exceptions are internal to the application and usually the application cannot anticipate or recover from such exceptions.
- These exceptions usually indicate programming errors, such as logical errors or improper use of an API.
- For example:
  - suppose a user specified the file name of the file to be opened.
- However, due to some logical error a `null` is passed to the application, then the application will throw a `NullPointerException`.
- The application can choose to catch this exception and display appropriate message to the user or eliminate the

Types of Errors and Exceptions: Runtime Exception

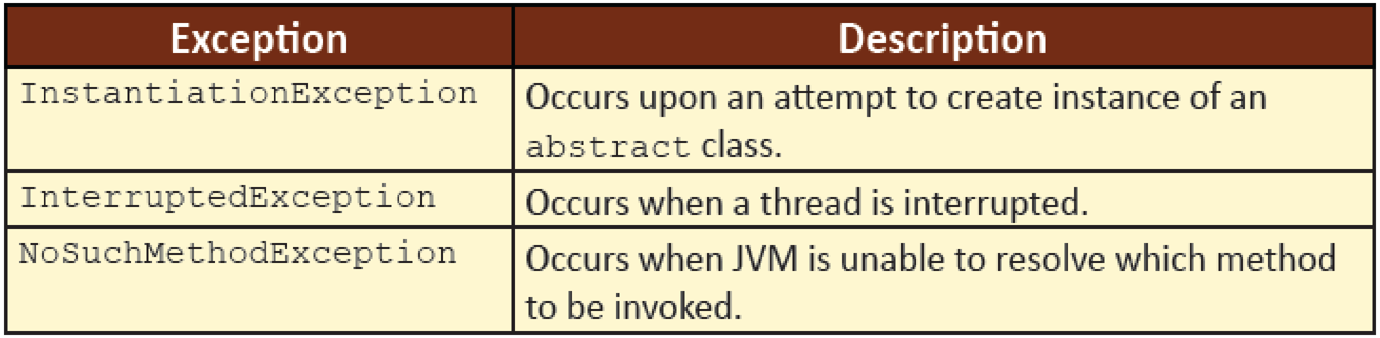
- Errors and runtime exceptions are collectively known as unchecked exceptions.
- In Java, `Object` class is the base class of the entire class hierarchy.
- `Throwable` class is the base class of all the exception classes.
- `Object` class is the base class of `Throwable`.
- `Throwable` class has two direct subclasses namely, `Exception` and `Error`.

Types of Errors and Exceptions



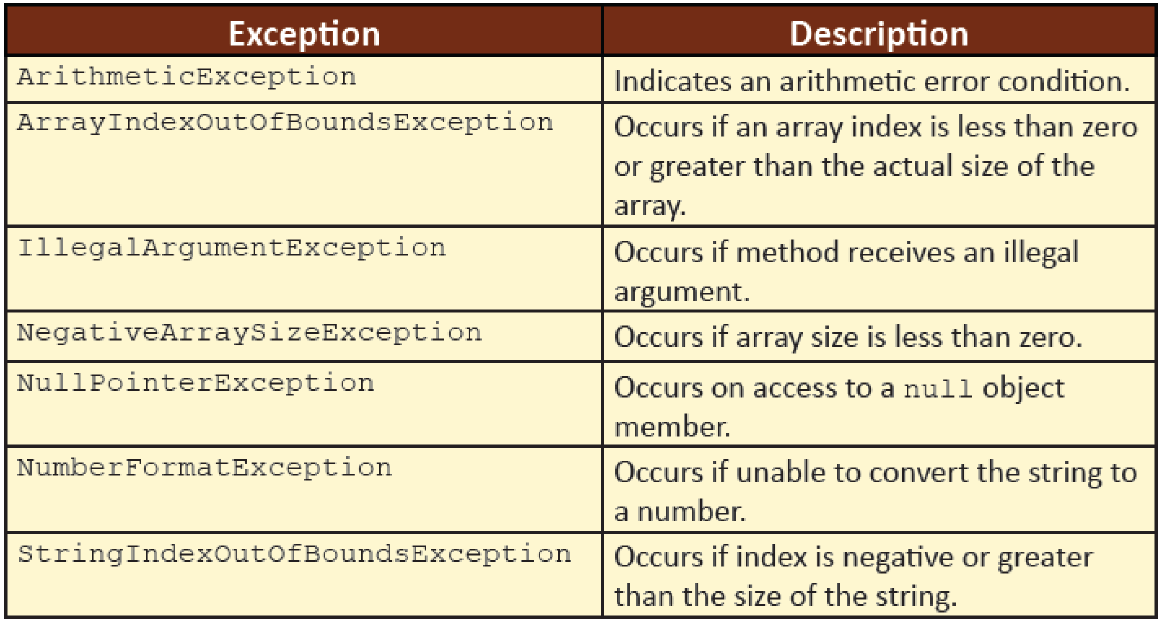
Types of Errors and Exceptions

- Following table lists some of the **checked** exceptions:



Exception	Description
`InstantiationException`	Occurs upon an attempt to create instance of an `abstract` class.
`InterruptedException`	Occurs when a thread is interrupted.
`NoSuchMethodException`	Occurs when JVM is unable to resolve which method to be invoked.

- Following table lists some of the commonly observed **unchecked** exceptions:



Exception	Description
`ArithmeticException`	Indicates an arithmetic error condition.
`ArrayIndexOutOfBoundsException`	Occurs if an array index is less than zero or greater than the actual size of the array.
`IllegalArgumentException`	Occurs if method receives an illegal argument.
`NegativeArraySizeException`	Occurs if array size is less than zero.
`NullPointerException`	Occurs on access to a `null` object member.
`NumberFormatException`	Occurs if unable to convert the string to a number.
`StringIndexOutOfBoundsException`	Occurs if index is negative or greater than the size of the string.

Exception Class

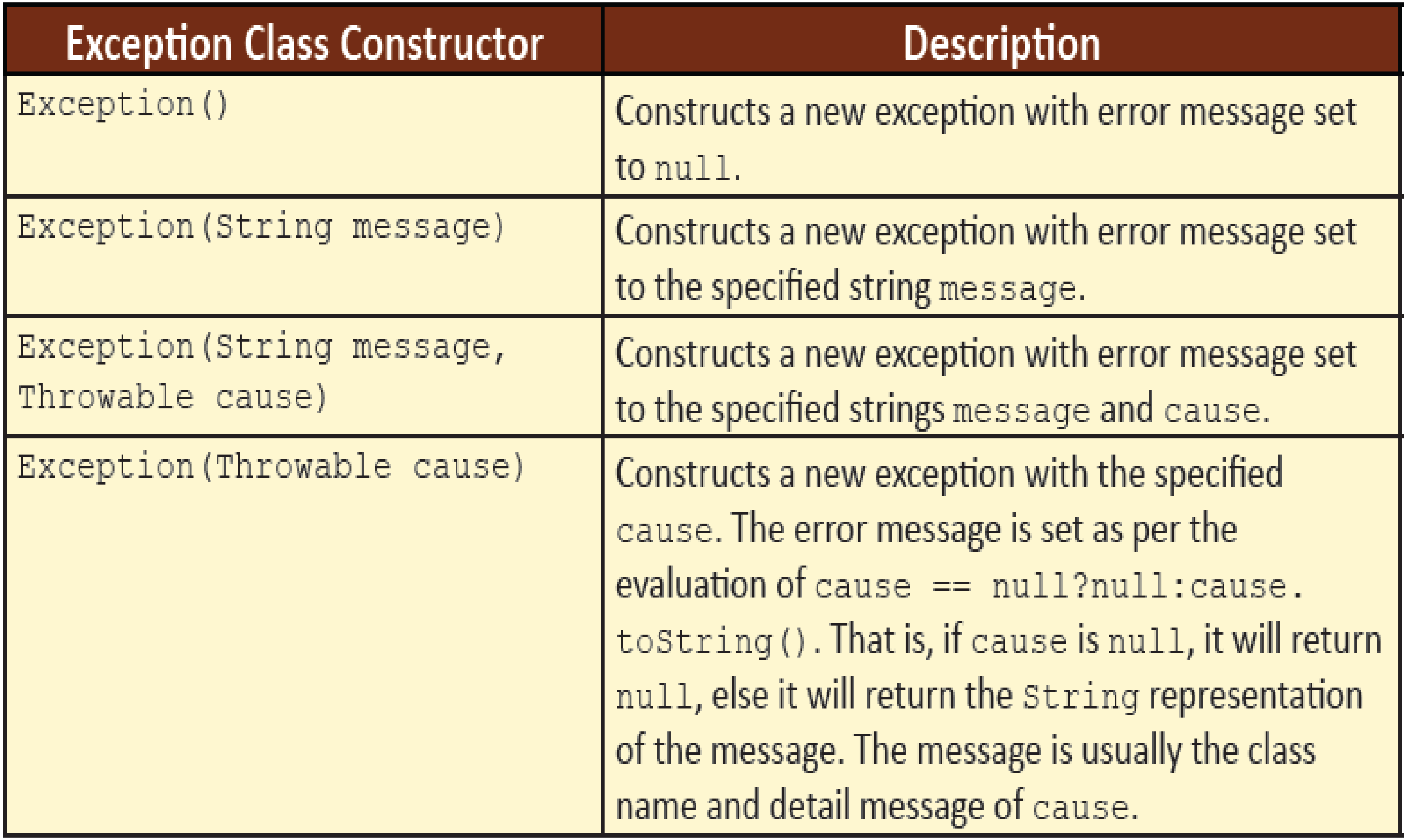
The class `Exception` and its subclasses indicate conditions that an application might attempt to handle.

The `Exception` class and all its subclasses except `RuntimeException` and its subclasses, are checked exceptions.

- The checked exceptions must be declared in a method or constructor's throws clause if the method or constructor is liable to throw the exception during its execution and propagate it further in the call stack.
- Following code snippet displays the structure of the Exception class:

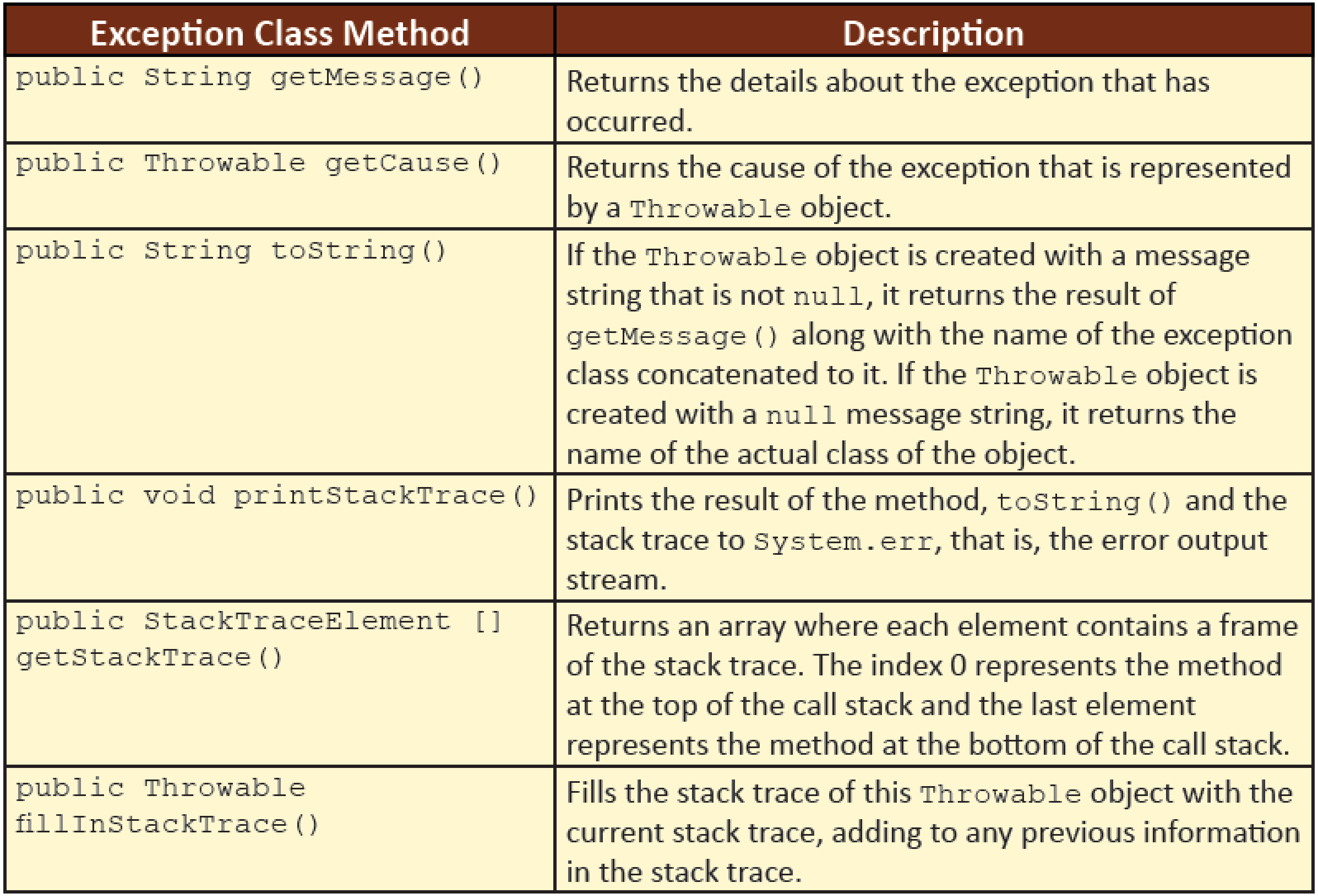
```
public class Exception extends Throwable{
   …
}
```

Exception Class



Exception Class Constructor	Description
Exception()	Constructs a new exception with error message set to null.
Exception(String message)	Constructs a new exception with error message set to the specified string message.
Exception(String message, Throwable cause)	Constructs a new exception with error message set to the specified strings message and cause.
Exception(Throwable cause)	Constructs a new exception with the specified cause. The error message is set as per the evaluation of cause == null?null:cause.toString(). That is, if cause is null, it will return null, else it will return the String representation of the message. The message is usually the class name and detail message of cause.

Exception Class



Exception Class Method	Description
`public String getMessage()`	Returns the details about the exception that has occurred.
`public Throwable getCause()`	Returns the cause of the exception that is represented by a `Throwable` object.
`public String toString()`	If the `Throwable` object is created with a message string that is not `null`, it returns the result of `getMessage()` along with the name of the exception class concatenated to it. If the `Throwable` object is created with a `null` message string, it returns the name of the actual class of the object.
`public void printStackTrace()`	Prints the result of the method, `toString()` and the stack trace to `System.err`, that is, the error output stream.
`public StackTraceElement [] getStackTrace()`	Returns an array where each element contains a frame of the stack trace. The index 0 represents the method at the top of the call stack and the last element represents the method at the bottom of the call stack.
`public Throwable fillInStackTrace()`	Fills the stack trace of this `Throwable` object with the current stack trace, adding to any previous information in the stack trace.

Handling Exceptions in Java

- Any exception that a method is liable to throw is considered to be as much a part of that method’s programming interface as its parameters and return value.
- The code that calls a method must be aware about the exceptions that a method may throw.
- This helps the caller to decide how to handle them if and when they occur.
- More than one runtime exceptions can occur anywhere in a program.

Handling Exceptions in Java

- Having to add code to handle runtime exceptions in every method declaration may reduce a program's clarity.
- Thus, the compiler does not require that a user must catch or specify runtime exceptions, although it does not object it either.

Handling Exceptions in Java

- A common situation where a user can throw a RuntimeException is when the user calls a method incorrectly.
- For example:
  - a method can check beforehand if one of its arguments is incorrectly specified as null.
  - In that case, the method may throw a NullPointerException, which is an unchecked exception.
- Thus, if a client is capable of reasonably recovering from an exception, make it a checked exception.
- If a client cannot do anything to recover from the exception, make it an unchecked exception.

try-catch Block

- The first step in creating an exception handler is to identify the code that may throw an exception and enclose it within the `try` block.
- The syntax for declaring a `try` block is as follows:

Syntax

```
try{
   // statement 1
   // statement 2
}
```

- The statements within the try block may throw an exception.
- Now, when the exception occurs, it is trapped by the try block and the runtime looks for a suitable handler to handle the exception.
- To handle the exception, the user must specify a catch block within the method that raised the exception or somewhere higher in the method call stack.

try-catch Block

- The syntax for declaring a `try-catch` block is as follows:

Syntax

```
try{
    // statements that may raise exception
    // statement 1
    // statement 2
}
catch(<exception-type> <object-name>){
    // handling exception
    // error message
}
```

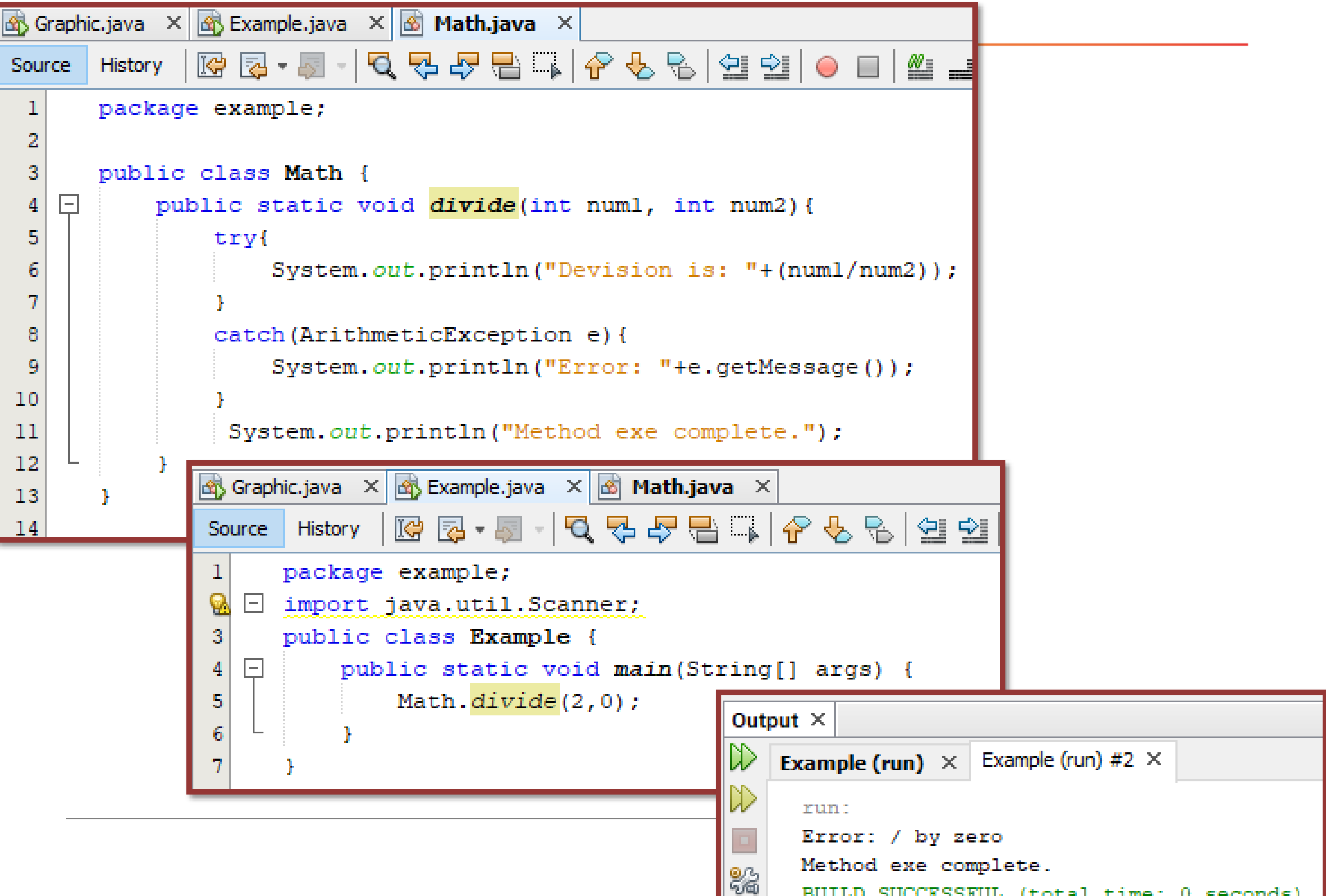
where,

exception-type: Indicates the type of exception that can be handled.

object-name: Object representing the type of exception.

- The catch block handles exceptions derived from Throwable class.

try-catch Block



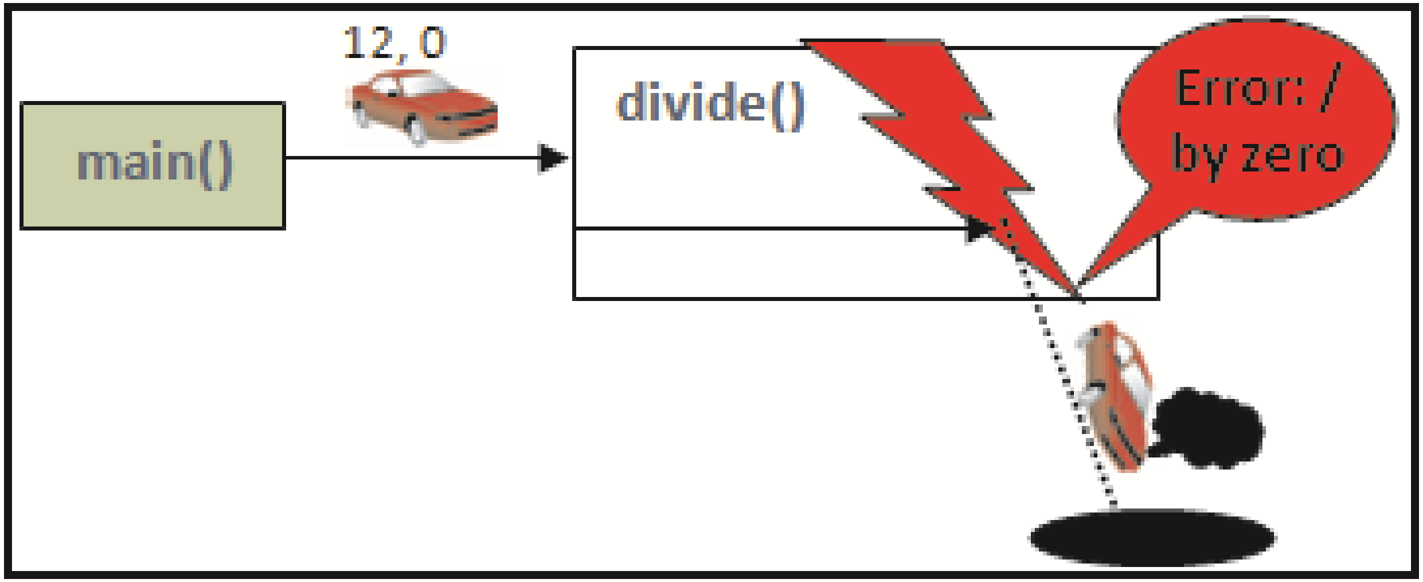
```java
package example;

public class Math {
    public static void divide(int num1, int num2){
        try{
            System.out.println("Devision is: "+(num1/num2));
        }
        catch(ArithmeticException e){
            System.out.println("Error: "+e.getMessage());
        }
         System.out.println("Method exe complete.");
    }
}
```

```java
package example;
import java.util.Scanner;
public class Example {
    public static void main(String[] args) {
        Math.divide(2,0);
    }
}
```

```
run:
Error: / by zero
Method exe complete.
BUILD SUCCESSFUL (total time: 0 seconds)
```

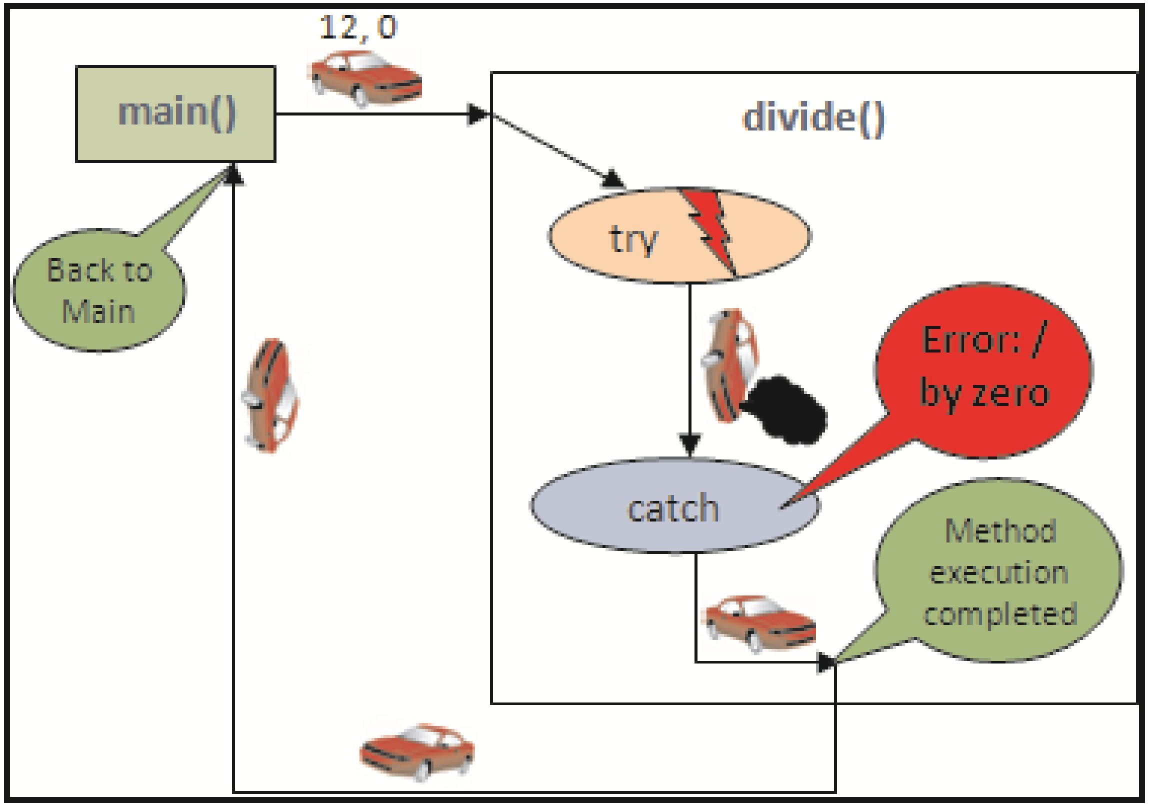
Execution Flow of Exceptions

- In the code, divide-by-zero exception occurs on execution of the statement `num1/num2`.
- If `try-catch` block is not provided, any code after this statement is not executed as an exception object is automatically created.
- Since, no `try-catch` block is present, JVM handles the exception, prints the stack trace, and the program is terminated.
- Following figure shows the execution of the code when `try-catch` block is not provided:



Execution Flow of Exceptions

- When the try-catch block is provided, the divide-by-zero exception occurring in the code is handled by the try-catch block and an exception message is displayed.
- Also, the rest of the code gets executed normally.
- Following figure shows the execution of the code when `try-catch` block is provided:



‘throw’ and ‘throws’ Keywords

Java provides the throw and throws keywords to explicitly raise an exception in the main() method.

The throw keyword throws the exception in a method.

The throws keyword indicates the exception that a method may throw.

The throw clause requires an argument of Throwable instance and raises checked or unchecked exceptions in a method.

‘throw’ and ‘throws’ Keywords

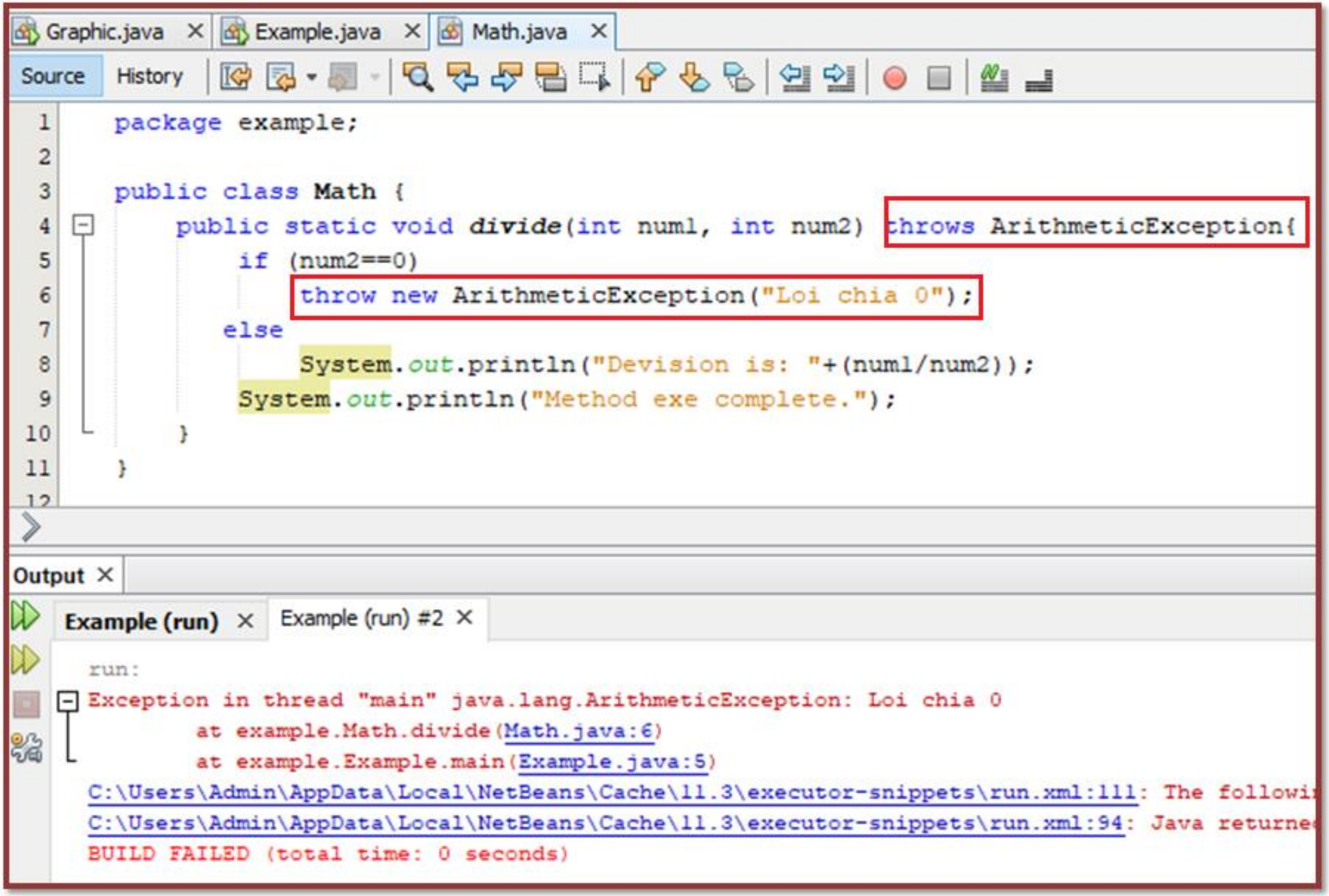
Java provides the throw and throws keywords to explicitly raise an exception in the main() method.

The throw keyword throws the exception in a method.

The throws keyword indicates the exception that a method may throw.

The throw clause requires an argument of Throwable instance and raises checked or unchecked exceptions in a method.

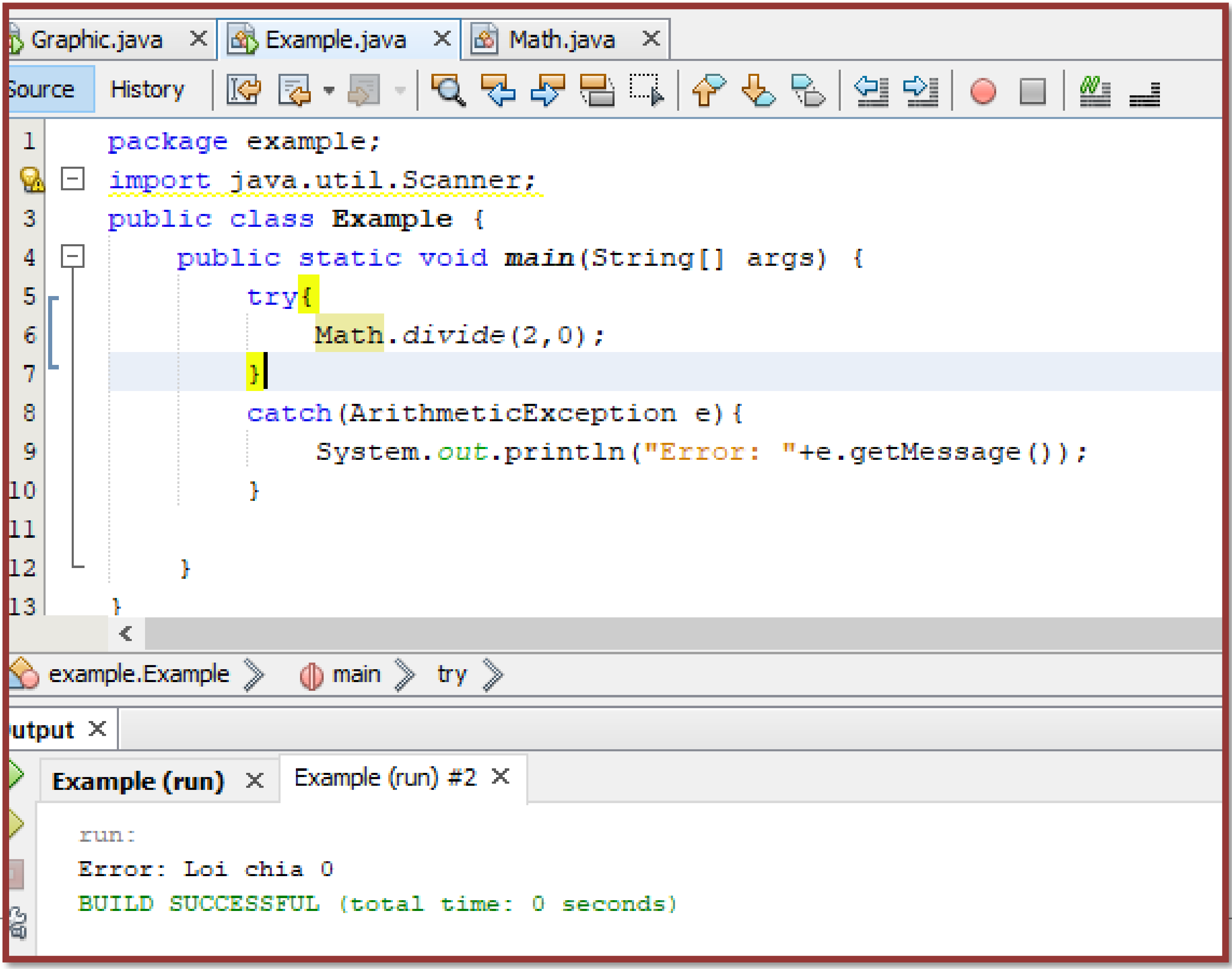
‘throw’ and ‘throws’ Keywords



```
package example;

public class Math {
    public static void divide(int num1, int num2) throws ArithmeticException{
        if (num2==0)
            throw new ArithmeticException("Loi chia 0");
       else
            System.out.println("Devision is: "+(num1/num2));
        System.out.println("Method exe complete.");
    }
}
```

```
run:
Exception in thread "main" java.lang.ArithmeticException: Loi chia 0
	at example.Math.divide(Math.java:6)
	at example.Example.main(Example.java:5)
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\run.xml:111: The followi
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\run.xml:94: Java returne
BUILD FAILED (total time: 0 seconds)
```

‘throw’ and ‘throws’ Keywords

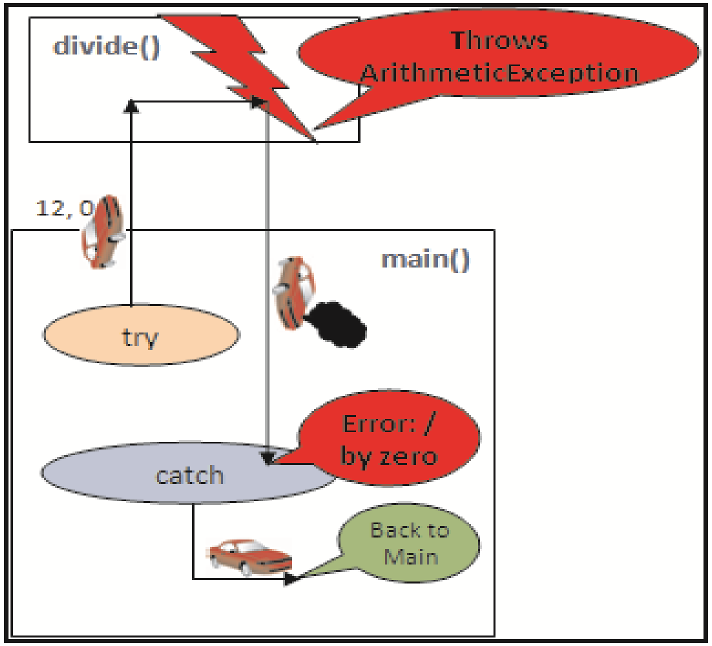


```
package example;
import java.util.Scanner;
public class Example {
    public static void main(String[] args) {
        try{
            Math.divide(2,0);
        }
        catch(ArithmeticException e){
            System.out.println("Error: "+e.getMessage());
        }

    }
}
```

```
run:
Error: Loi chia 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

‘throw’ and ‘throws’ Keywords

- The control returns back to the caller, that is, the `main()` method where it is finally handled.
- The `catch` block was executed and the result of `getMessage()` is displayed to the user.
- Notice, that the remaining statement of the **`divide(int,int)`** method is not executed in this case.
- Following figure shows the execution of the code when `throw` and `throws` clauses are used:



Multiple ‘catch’ Blocks

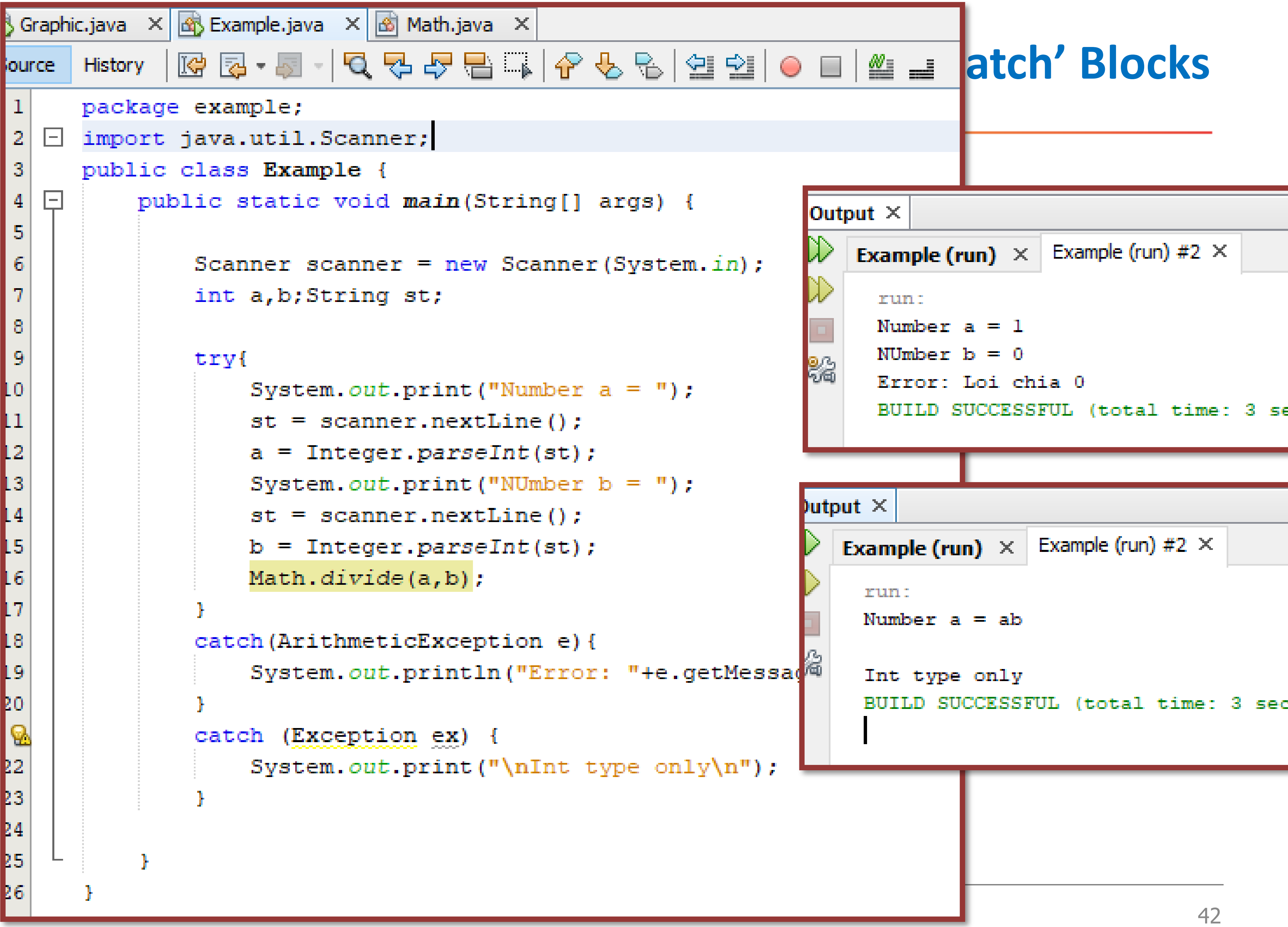
- The user can associate multiple exception handlers with a `try` block by providing more than one `catch` blocks directly after the `try` block.
- The syntax for declaring a `try` block with multiple `catch` blocks is as follows:

Syntax

```
try
{…}
catch (<exception-type> <object-name>)
{…}
catch (<exception-type> <object-name>)
{…}
```

- In this case, each catch block is an exception handler that handles a specific type of exception indicated by its argument exception-type.
- The runtime system invokes the handler in the call stack whose exception-type matches the type of the exception thrown.

atch' Blocks

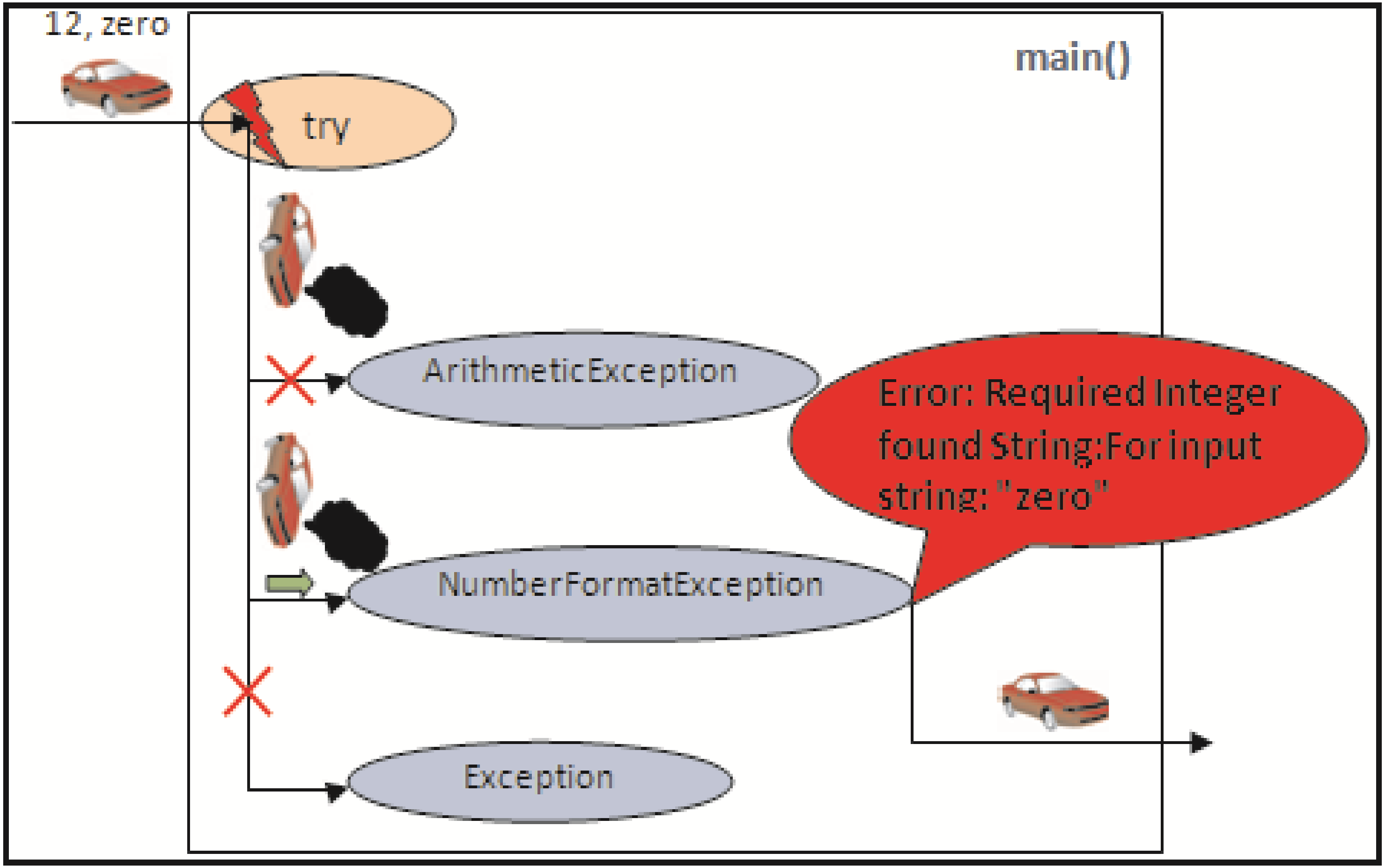


```java
package example;
import java.util.Scanner;
public class Example {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        int a,b;String st;

        try{
            System.out.print("Number a = ");
            st = scanner.nextLine();
            a = Integer.parseInt(st);
            System.out.print("NUmber b = ");
            st = scanner.nextLine();
            b = Integer.parseInt(st);
            Math.divide(a,b);
        }
        catch(ArithmeticException e){
            System.out.println("Error: "+e.getMessa
        }
        catch (Exception ex) {
            System.out.print("\nInt type only\n");
        }


    }
}
```

```
run:
Number a = 1
NUmber b = 0
Error: Loi chia 0
BUILD SUCCESSFUL (total time: 3 se
```

```
run:
Number a = ab

Int type only
BUILD SUCCESSFUL (total time: 3 sec
```

Multiple ‘catch’ Blocks



‘finally’ Block

Java provides the `finally` block to ensure execution of certain statements even when an exception occurs.

The `finally` block is always executed irrespective of whether or not an exception occurs in the try block.

This ensures that the cleanup code is not accidentally bypassed by a `return`, `break`, or `continue` statement.

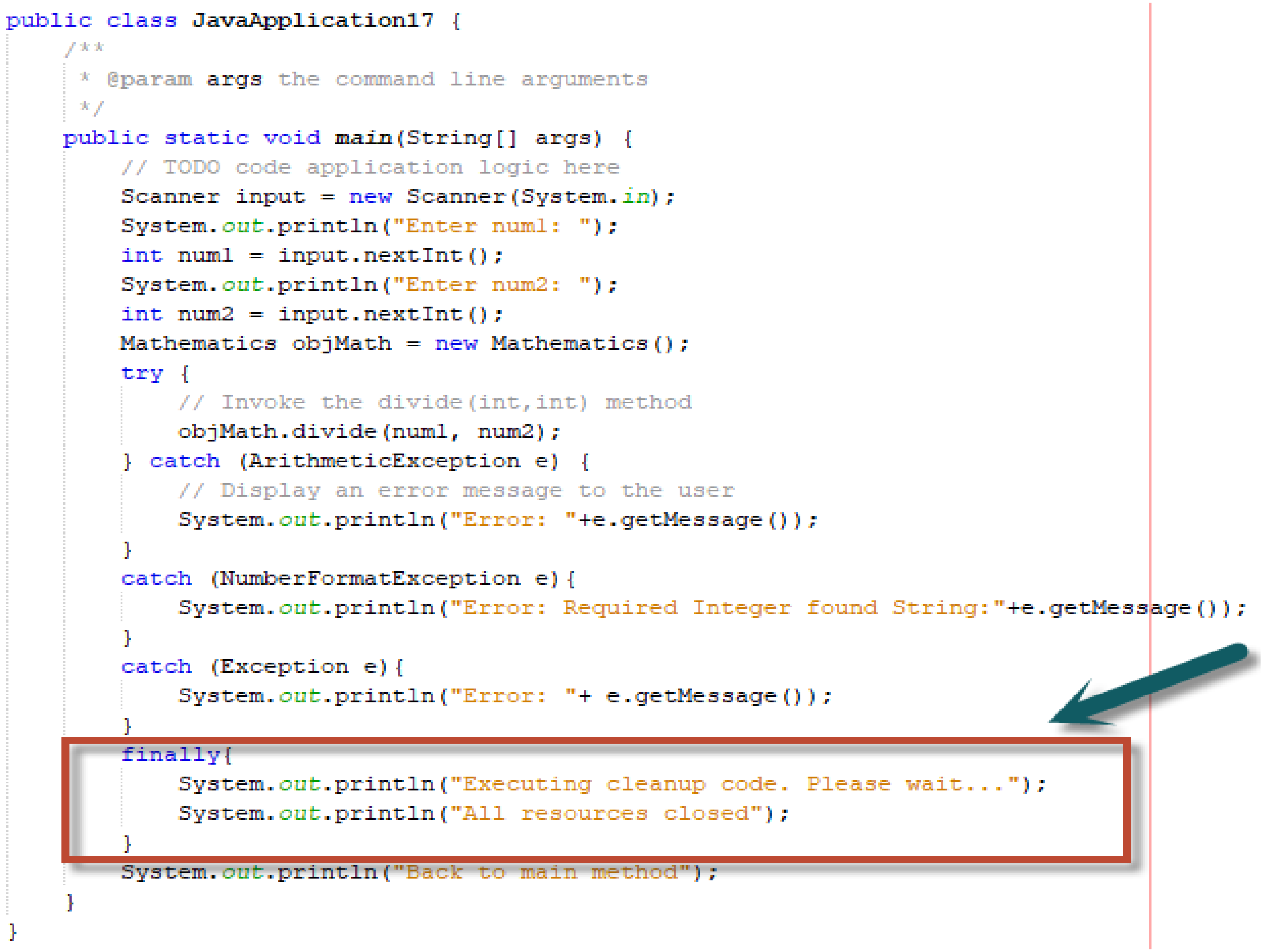
The `finally` block is mainly used as a tool to prevent resource leaks.

‘finally’ Block

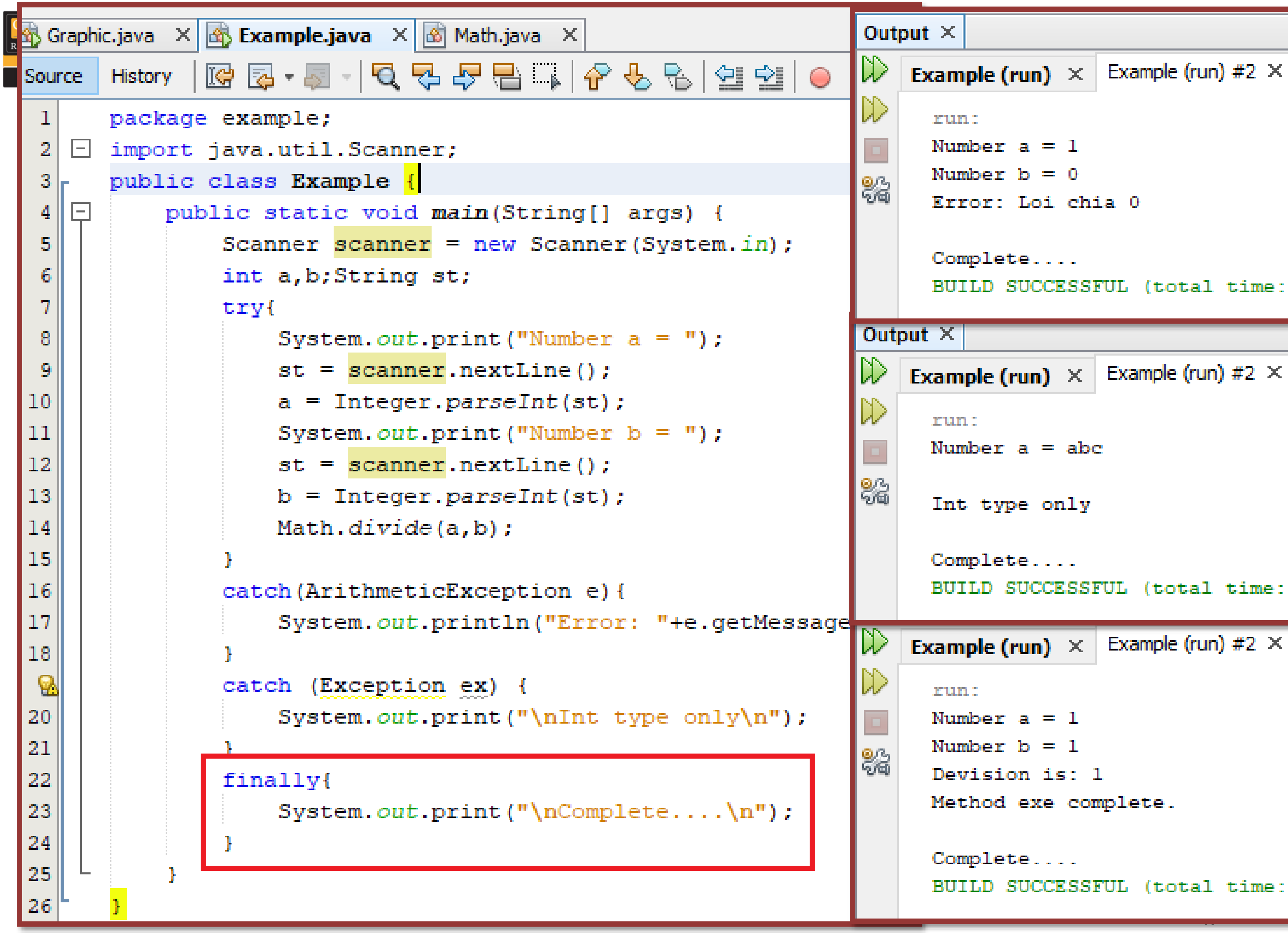
Syntax

```
try{
   // statements that may raise exception
   // statement 1
   // statement 2
}
catch(<exception-type> <object-name>){
   // handling exception
   // error message
}
finally{
   // clean-up code
   // statement 1
   // statement 2
}
```

‘finally’ Block



```
public class JavaApplication17 {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        Scanner input = new Scanner(System.in);
        System.out.println("Enter num1: ");
        int num1 = input.nextInt();
        System.out.println("Enter num2: ");
        int num2 = input.nextInt();
        Mathematics objMath = new Mathematics();
        try {
            // Invoke the divide(int,int) method
            objMath.divide(num1, num2);
        } catch (ArithmeticException e) {
            // Display an error message to the user
            System.out.println("Error: "+e.getMessage());
        }
        catch (NumberFormatException e){
            System.out.println("Error: Required Integer found String:"+e.getMessage());
        }
        catch (Exception e){
            System.out.println("Error: "+ e.getMessage());
        }
        finally{
            System.out.println("Executing cleanup code. Please wait...");
            System.out.println("All resources closed");
        }
        System.out.println("Back to main method");
    }
}
```



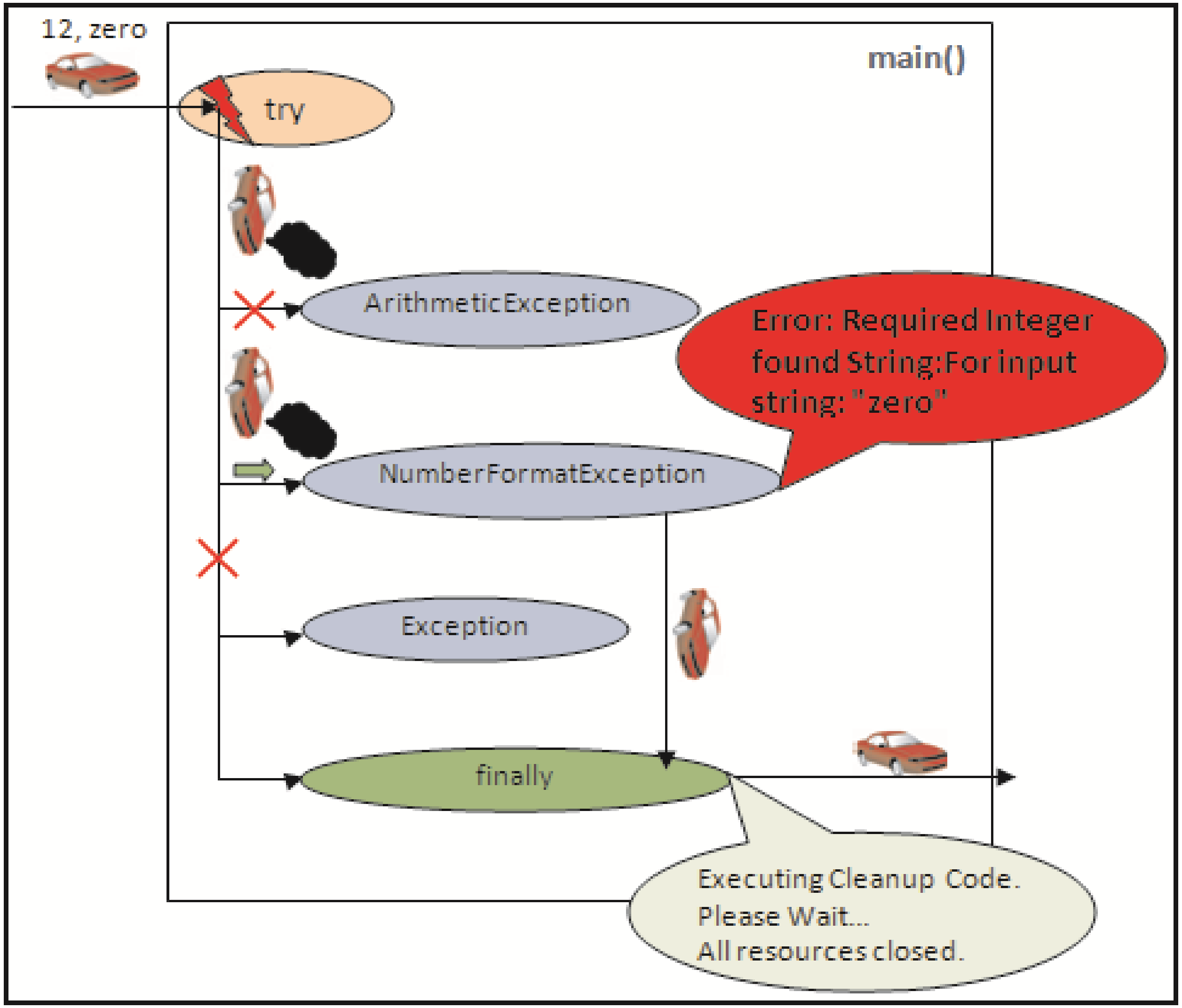
```java
package example;
import java.util.Scanner;
public class Example {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int a,b;String st;
        try{
            System.out.print("Number a = ");
            st = scanner.nextLine();
            a = Integer.parseInt(st);
            System.out.print("Number b = ");
            st = scanner.nextLine();
            b = Integer.parseInt(st);
            Math.divide(a,b);
        }
        catch(ArithmeticException e){
            System.out.println("Error: "+e.getMessage
        }
        catch (Exception ex) {
            System.out.print("\nInt type only\n");
        }
        finally{
            System.out.print("\nComplete....\n");
        }
    }
}
```

```
run:
Number a = 1
Number b = 0
Error: Loi chia 0

Complete....
BUILD SUCCESSFUL (total time:
```

```
run:
Number a = abc

Int type only

Complete....
BUILD SUCCESSFUL (total time:
```

```
run:
Number a = 1
Number b = 1
Devision is: 1
Method exe complete.

Complete....
BUILD SUCCESSFUL (total time:
```

‘finally’ Block


12, zero
main()
try
ArithmeticException
Error: Required Integer found String:For input string: "zero"
NumberFormatException
Exception
finally
Executing Cleanup Code. Please Wait... All resources closed.

User-defined Exceptions

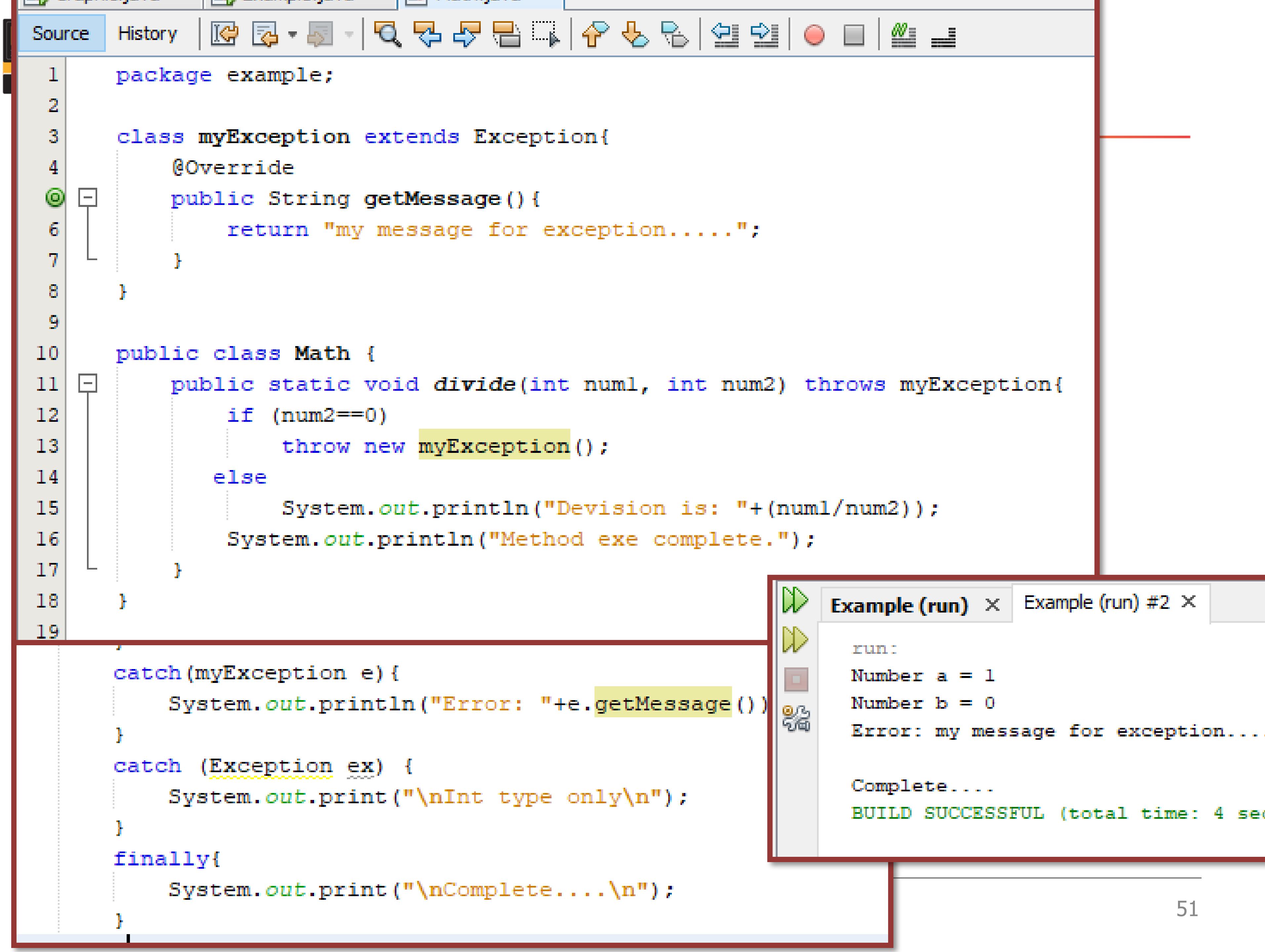
One can create a custom exception class when:

- The built-in exception type does not fulfill the requirement.
- It is required to differentiate your exceptions from those thrown by classes written by other vendors.
- The code throws more than one related exception.

Creating a User-defined Exception

Syntax

```
public class <ExceptionName> extends Exception {

}
```



```
package example;

class myException extends Exception{
    @Override
    public String getMessage(){
        return "my message for exception.....";
    }
}

public class Math {
    public static void divide(int num1, int num2) throws myException{
        if (num2==0)
            throw new myException();
       else
            System.out.println("Devision is: "+(num1/num2));
        System.out.println("Method exe complete.");
    }
}
```

```
}
catch(myException e){
    System.out.println("Error: "+e.getMessage())
}
catch (Exception ex) {
    System.out.print("\nInt type only\n");
}
finally{
    System.out.print("\nComplete....\n");
}
```

```
run:
Number a = 1
Number b = 0
Error: my message for exception...

Complete....
BUILD SUCCESSFUL (total time: 4 se
```

Student Activities

- Modify code of Student Class: Alert if
  - Name is not text
  - Year_of_birth is not int
  - Gender is not F or M
  - GPA is not float

Assertions

- A statement in Java that allows the programmer to test his/her assumptions about the program.
- Each assertion is composed of a boolean expression that is believed to be true when the assertion executes, if not, the system will throw an error.
- By verifying that the boolean expression is indeed true, the assertion confirms the assumptions about the behavior of the program.

➡ This helps to increase the programmer’s confidence that the code is free of errors.

Assertions

- The syntax of assertion statement has the following two forms:

Syntax

```
assert <boolean_expression>;
```

Syntax

```
assert <boolean_expression> : <detail_expression> ;
```

- This version of the `assert` statement is used to provide a detailed message for the `AssertionError`.
- The system will pass the value of `detail_expression` to the appropriate `AssertionError` constructor.
- The constructor uses the string representation of the value as the error's detail message.

Assertions

- To ensure that assertions do not become a performance liability in deployed applications, assertions can be enabled or disabled when the program is started.
- Assertions are disabled by default.
- Disabling assertions removes their performance related issues entirely.
- Once disabled, they become empty statements in the code semantics.
- Assertion checking is disabled by default.
- Assertions can be enabled at command line by using the following command:

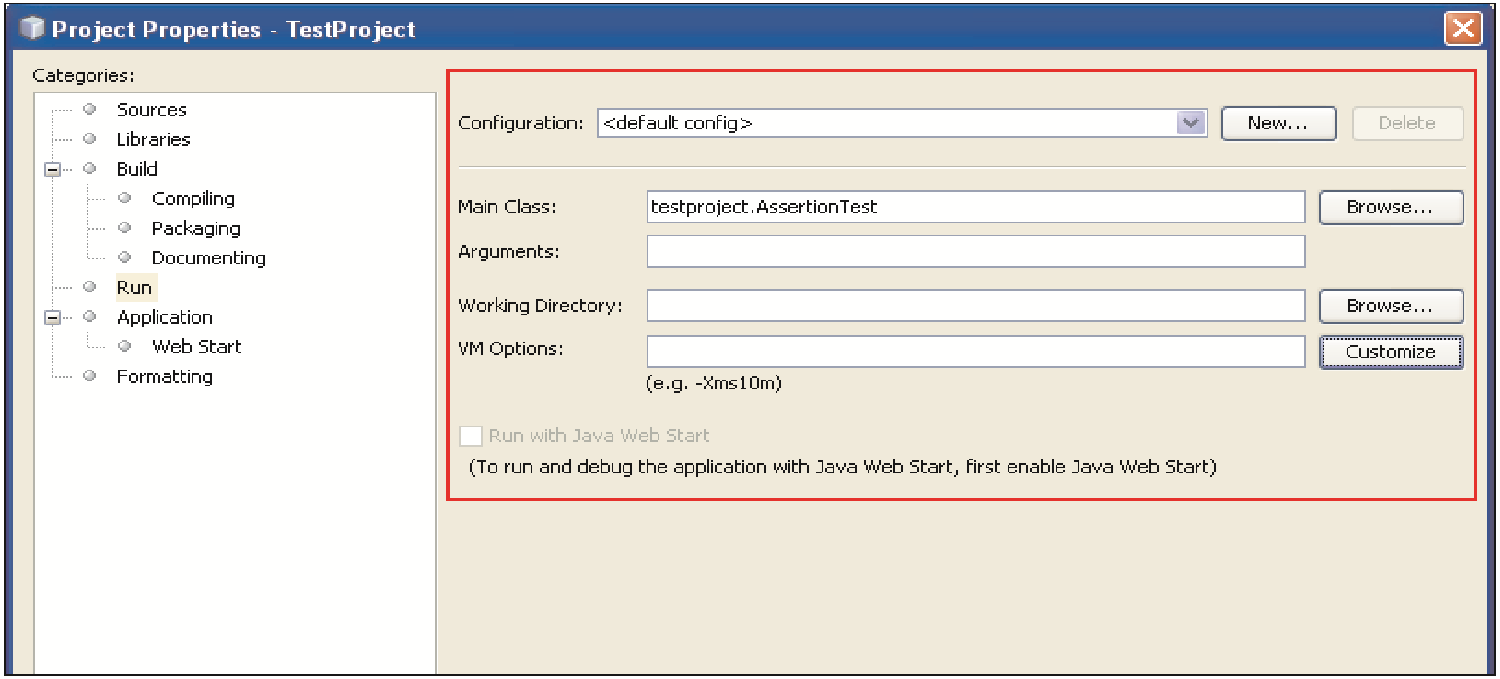
java –ea <class-name> or

java –enableassertions <class-name>

Assertions

- To enable assertions in NetBeans IDE, perform the following steps:

1. Right-click the project in the Projects tab. A pop-up menu appears.
2. Select Properties. The Project Properties dialog box is displayed.
3. Select Run from the Categories pane. The runtime settings pane is displayed on the right.



Project Properties - TestProject

Categories:

- Sources
- Libraries
- Build
  - Compiling
  - Packaging
  - Documenting
- Run
- Application
  - Web Start
- Formatting

Configuration: <default config>

Main Class: testproject.AssertionTest

Arguments:

Working Directory:

VM Options:

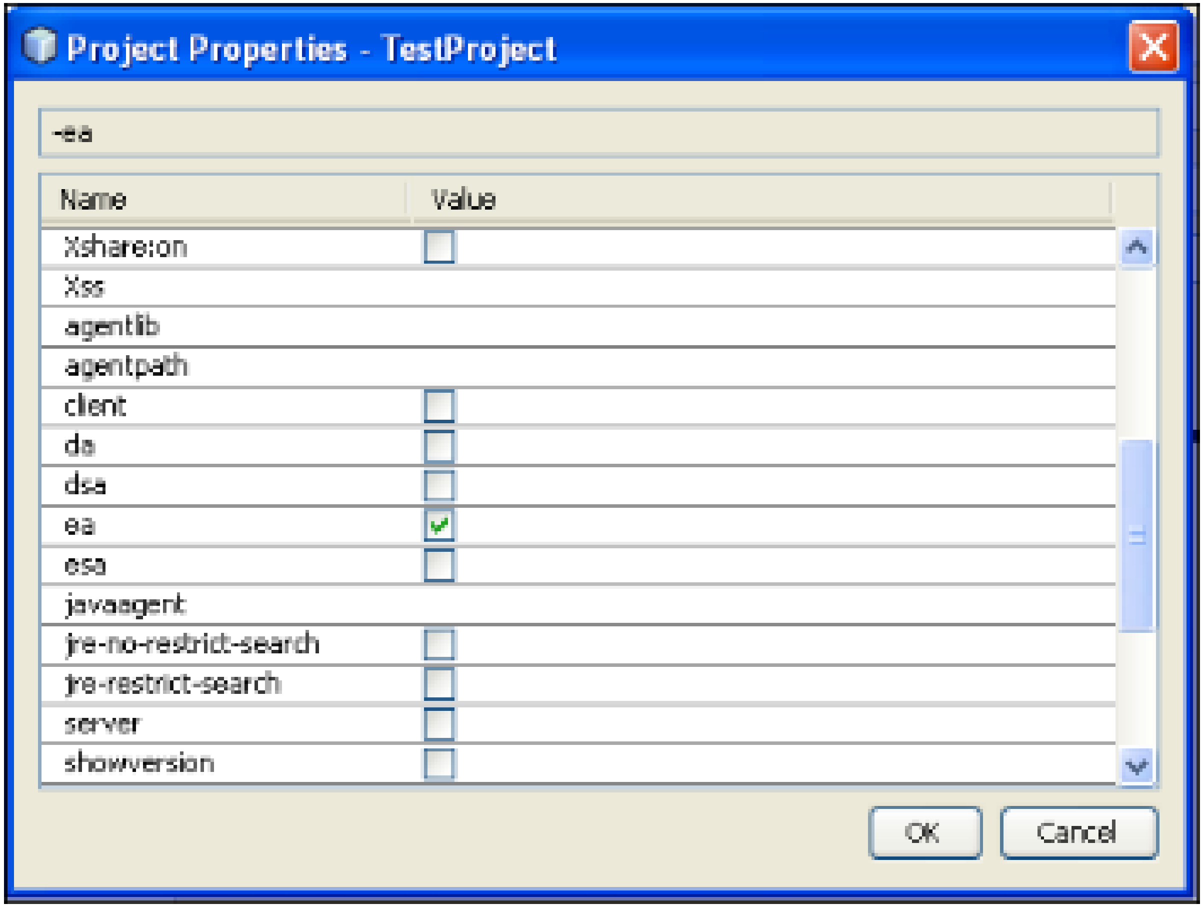
(e.g. -Xms10m)

□ Run with Java Web Start

(To run and debug the application with Java Web Start, first enable Java Web Start)

Assertions

4. Click the ***Customize*** button. The ***Project Properties*** dialog box is displayed.
5. Scroll down and select the ***ea*** checkbox.



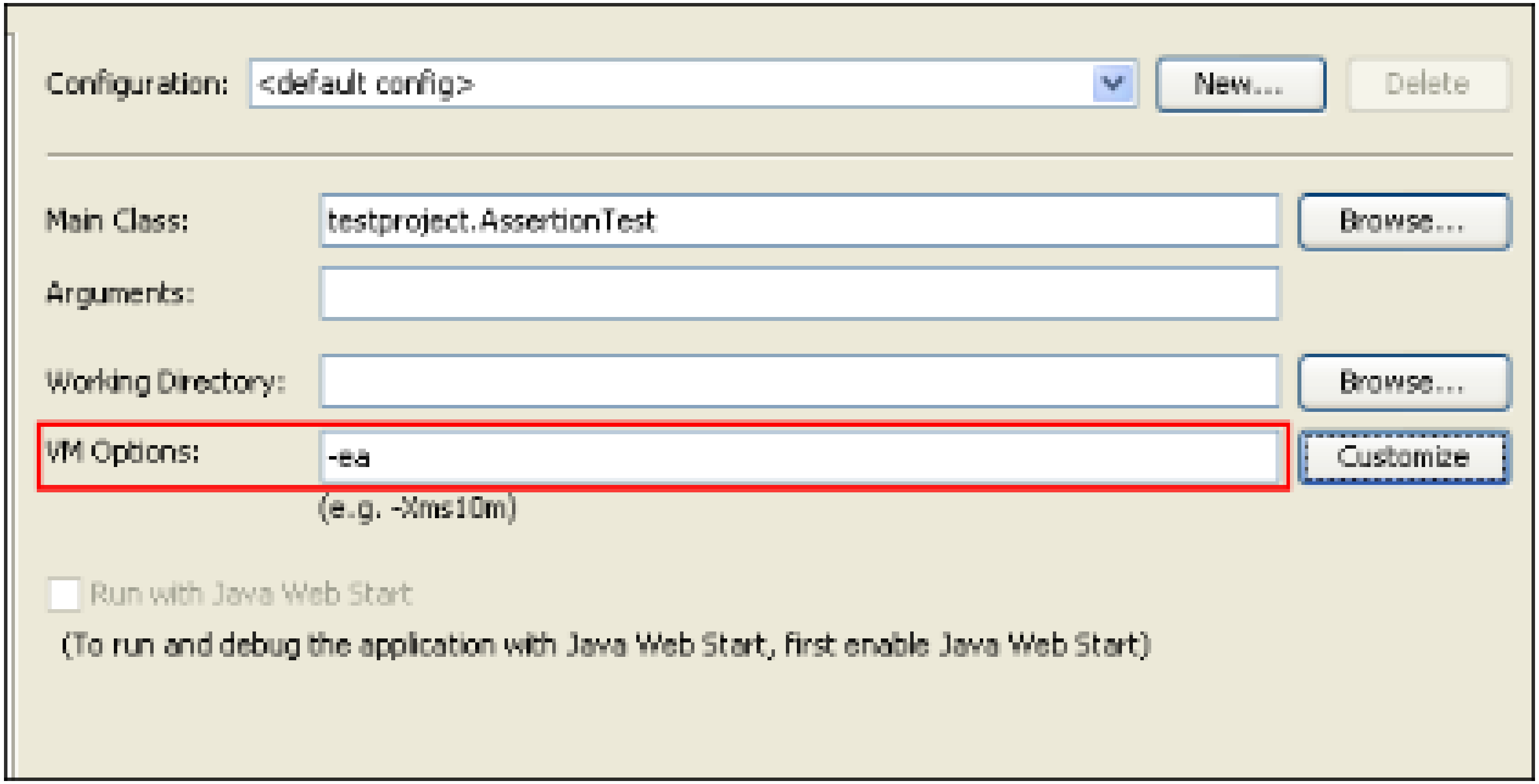
Project Properties - TestProject

-ea

Name	Value
Xshare:on	☐
Xss	
agentlib	
agentpath	
client	☐
da	☐
dsa	☐
ea	☑
esa	☐
javaagent	
jre-no-restrict-search	☐
jre-restrict-search	☐
server	☐
showversion	☐

Assertions

6. Click **OK**. The **–ea** option is set in the **VM Options** text box.



Configuration: <default config>

Main Class: testproject.AssertionTest

Arguments:

Working Directory:

VM Options: -ea

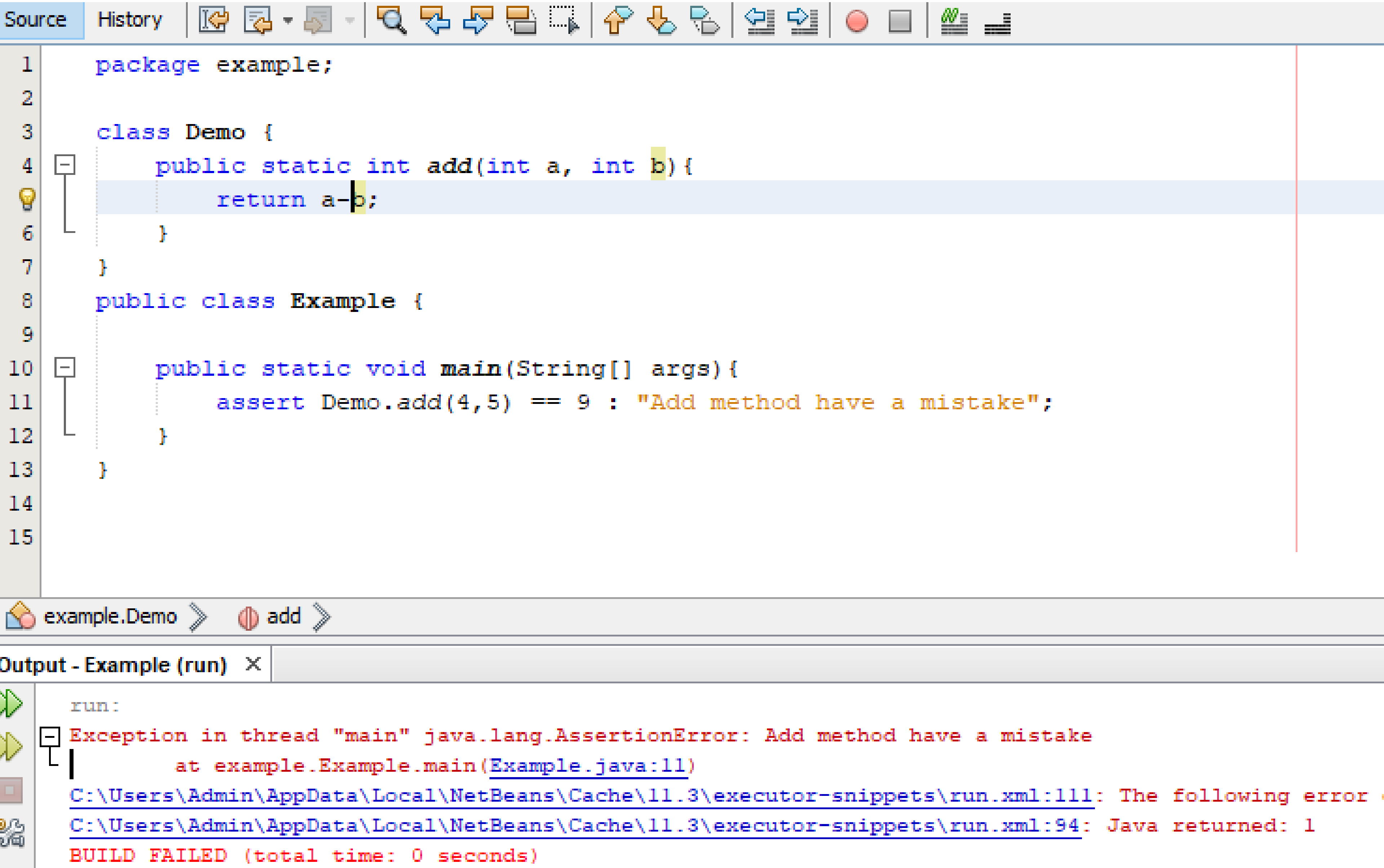
(e.g. -Xms10m)

□ Run with Java Web Start

(To run and debug the application with Java Web Start, first enable Java Web Start)

7. Click ***OK***.

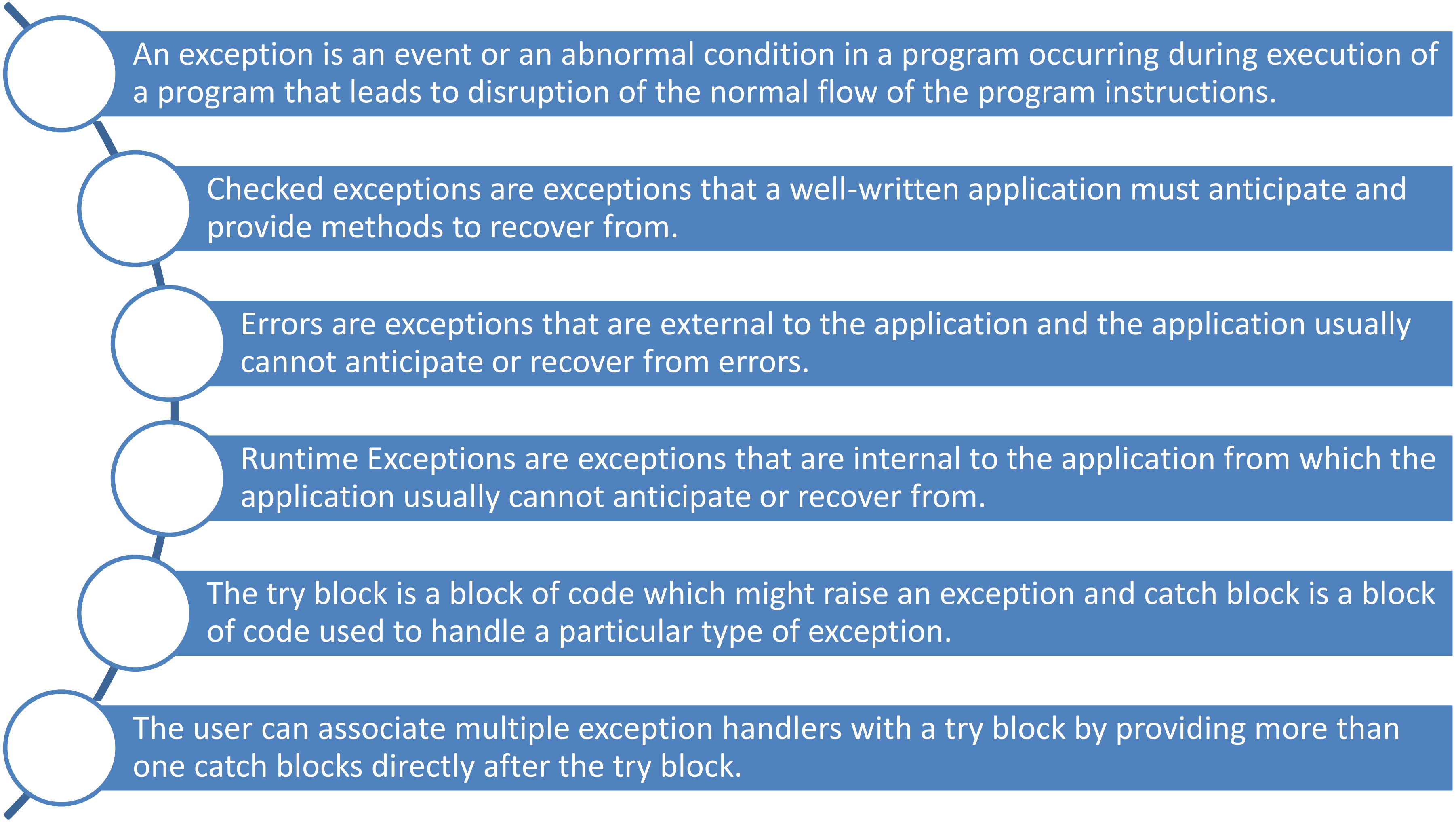
Assertions



```java
package example;

class Demo {
    public static int add(int a, int b){
        return a-b;
    }
}
public class Example {

    public static void main(String[] args){
        assert Demo.add(4,5) == 9 : "Add method have a mistake";
    }
}
```

```
run:
Exception in thread "main" java.lang.AssertionError: Add method have a mistake
	at example.Example.main(Example.java:11)
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\run.xml:111: The following error
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\run.xml:94: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

Summary



An exception is an event or an abnormal condition in a program occurring during execution of a program that leads to disruption of the normal flow of the program instructions.

Checked exceptions are exceptions that a well-written application must anticipate and provide methods to recover from.

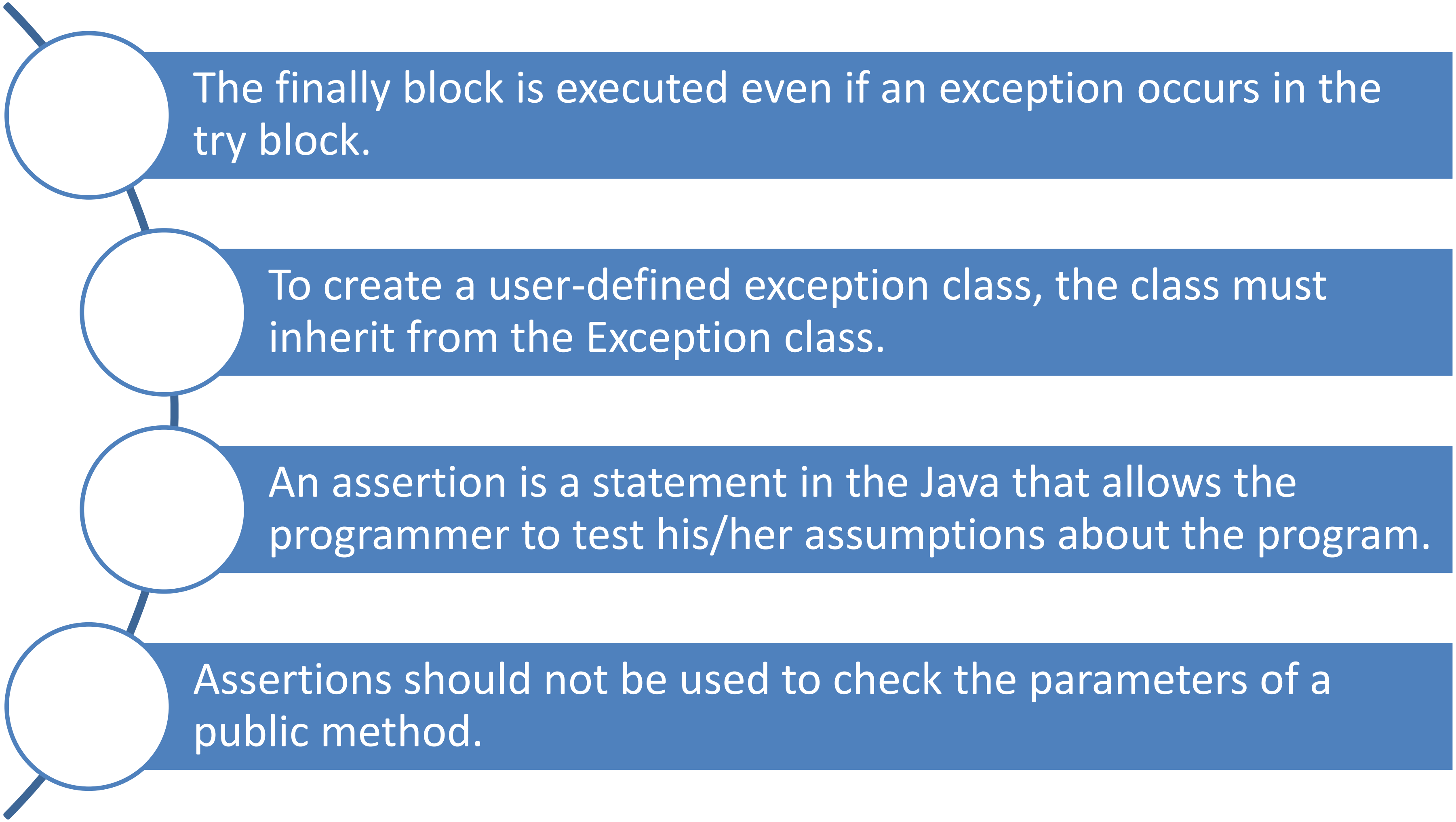
Errors are exceptions that are external to the application and the application usually cannot anticipate or recover from errors.

Runtime Exceptions are exceptions that are internal to the application from which the application usually cannot anticipate or recover from.

The try block is a block of code which might raise an exception and catch block is a block of code used to handle a particular type of exception.

The user can associate multiple exception handlers with a try block by providing more than one catch blocks directly after the try block.

Summary



The finally block is executed even if an exception occurs in the try block.

To create a user-defined exception class, the class must inherit from the Exception class.

An assertion is a statement in the Java that allows the programmer to test his/her assumptions about the program.

Assertions should not be used to check the parameters of a public method.

THANK YOU!