Object-oriented Programming in Java

Session: 1

Exceptions and Assertions

Objectives

- Describe exception
- Explain the use of try-catch-finally blocks
- Explain throwing and handling exceptions
- Explain handling multiple exceptions in a single catch block
- Explain the use of try-with-resources
- Describe creation and use of custom exceptions
- Explain assertions and its types

Introduction

- Java programming language provides the ability to handle unexpected events in the program.
- This is because, even though a code is well written, it is prone to behave erroneously.
- With the help of exception handling, the developer can ensure that the user gets a proper message in case some error occurs in the program.
- It also helps to ensure that in case of an error, the data and processes that the program is using do not get affected adversely.

Overview of Exceptions

- An exception is an event occurring during program execution that leads to disruption of the normal flow of the program's instructions.
- Exception handling in Java is a way for ensuring smooth execution of a program.
- To handle the exceptions, Java provides the `try-catch-finally` blocks.
- Using these blocks, the developer can check the program statements for errors and handle them in case they occur.

`try-catch-finally` Block [1-5]

- In order to handle exceptions, the developer needs to identify the statements that may lead to exceptions and enclose them within a try block.
- Next, exception handlers need to be associated with a `try` block by providing one or more `catch` blocks directly after the `try` block.
- The syntax of a `try-catch` block is as follows:

Syntax

```
try {
        . . .
       } catch (ExceptionType name) {
       } catch (ExceptionType name) {
}
. . .
```

try-catch-finally Block [2-5]

The following Code Snippet shows an example of `try-catch` block:

Code Snippet

```
public class Calculator{
 int result;
 public void divide(int num1, int num2){
   try{
        result = num1/num2; //line 1 -- statement that might
raise exception
       }catch(ArithmeticException ex){
   // printing the error message
    System.out.println("Denominator cannot be set to zero!!!"
+ ex.getMessage());
       }
   }
}
```

`try-catch-finally` Block [3-5]

- The `finally` block is executed even if an exception occurs in the `try` block.
- It helps the programmer to ensure that the cleanup code does not get accidentally bypassed by a `break`, `continue`, or `return` statement in the `try` block.
- It is advisable to write the cleanup code in a `finally` block, even when no exceptions are expected.
- Some conditions in which the `finally` block may not execute are as follows:
  - When the Java Virtual Machine (JVM) exits while the try or catch code is being executed.
  - If the thread executing the try or catch code, it is interrupted or killed.
- In these cases, the `finally` block may not execute even though the application as a whole continue execution.

`try-catch-finally` Block [4-5]

The following Code Snippet explains the use of the `finally` block:

Code Snippet

```
. . .
PrintWriter objPwOut = null; // PrintWriter object
 public void writeToFile{
  try {
      // initializing the PrintWriter with a file
name
objPwOut = new PrintWriter(“C:\\MyFile.txt”);
   } catch (FileNotFoundException ex) {
      // printing the error message
         System.out.println(“File Does not Exist “ +
ex.getMessage());
   } finally {
       // verifying if the PrintWriter object is still
open
```

try-catch-finally Block [5-5]

```
             if (objPwOut != null) {
                // closing the PrintWriter object
                   objPwOut.close();
                   System.out.println("PrintWriter 
closed");
             }
    }
}
```

throw and throws Keywords [1-6]

- At times, it might be required to let a method transfer the control further up the call stack to handle the exception.
- For example, while making connection to servers, one might not be able to anticipate the type of format in which the server id will be provided by the user.
- In this case, it is advisable not to catch the exception and to allow a method further up the call stack to handle it.
- If a method does not catch the checked exceptions that can occur within it, it must specify that it can throw these exceptions.
- The `throws` clause is written after the method name and argument list and before the opening brace of the method.

throw and throws Keywords [2-6]

The following Code Snippet shows the modified `writeToFile()` method with the `throws` clause:

Code Snippet

```
. . .
PrintWriter objPwOut = null;
 public void writeToFile throws FileNotFoundException{
  try {
       objPwOut = new PrintWriter("C:\\MyFile.txt");
  } finally {
       if (objPwOut != null) {
            objPwOut.close();
            System.out.println("PrintWriter closed");
       }
  }
}
```

`throw` and `throws` Keywords [3-6]

- The Java platform provides several exception classes which are descendants of the `Throwable` class.
- These classes allow programs to differentiate among the different types of exceptions that can occur during the execution of a program.
- All methods use the `throw` statement to throw an exception.
- The `throw` statement takes a single argument which is a `Throwable` object.
- Throwable objects are instances of any subclass of the `Throwable` class.

throw and throws Keywords [4-6]

In the following Code Snippet, the `writeToFile()` method is made to throw the `FileNotFoundException`:

Code Snippet

```
import java.io.*;
public class FileWriting {
    PrintWriter objPwOut = null;
    // Declares the exception in the throws clause
   public void writeToFile() throws
FileNotFoundException
    {
      try {
           objPwOut = new PrintWriter(“C:\\MyFile.txt”);
      } catch (FileNotFoundException ex) {
           // Re-throwing the exception
            throw new FileNotFoundException();
      } finally {
```

throw and throws Keywords [5-6]

```
              if (objPwOut != null) {
                   objPwOut.close();
                   System.out.println("PrintWriter
closed"); }
        }
   }
      public static void main(String[] args)
      {
       try{
            FileWriting fw = new FileWriting();
            fw.writeToFile();
       } catch(FileNotFoundException ex)
```

throw and throws Keywords [6-6]

```
         {
           // Catching the exception
              System.out.println("File does not Exist "
+ ex.getMessage()); }
         }
    }
}
```

In the code:

- The `FileNotFoundException` is caught by the `catch` block of the `writeToFile()` method.
- However, the exception is not handled within this block, but is thrown back using the `throw new FileNotFoundException()` statement.
- Thus, the exception is transferred further up in the call stack. Now, the caller of the `writeToFile()` method will have to handle this exception.
- Since, `main()` is the caller method, the exception will have to be handled by `main()`.
- Hence, a `catch` block for `FileNotFoundException` has been provided in the `main()` method where the exception will be handled.

Throwing Exceptions from Methods [1-2]

- One can directly throw exceptions from a method and transfer it to a method higher up in the hierarchy.
- Consider the following Code Snippet:

Code Snippet

```
public class Calculator {
 public void divide(int a, int b) throws
ArithmeticException
 {
  if(b==0){
     throw new ArithmeticException(); // throwing
exception
   }
    int result = a/b;
    System.out.println("Result is " + result);
 }
}
```

Throwing Exceptions from Methods [2-2]

```
public class TestCalculator {
 public static void main(String[] args) {
   try{
        Calculator objCalc = new Calculator();
        // Invoking the divide() method
          objCalc.divide(Integer.parseInt(args[0]),
Integer. parseInt(args[1]));
   }catch(ArithmeticException ex){
        System.out.println("Denominator cannot be set to
zero");
   }
  }
}
```

Handling Multiple Exceptions in a Single `catch` Block [1-3]

- Java SE 7 and later versions provide the feature of handling more than one exception in a single `catch` block.
- This feature helps to reduce code duplication and prevent the use of a much generalized exception.
- To create a multiple exception `catch` block, specify the types of exceptions that `catch` block can handle separated by a vertical bar (`|`) as follows:

Syntax

```
catch (ExceptionType1|ExceptionType2 ex) {
// statements
}
```

- Since, the `catch` block handles more than one type of exception, then the catch parameter is implicitly `final`.
- Therefore, one cannot assign any values to it within the `catch` block.

Handling Multiple Exceptions in a Single catch Block [2-3]

The following Code Snippet shows an example of handling multiple exceptions in a single `catch` block:

Code Snippet

```
public class Calculator {
 public static void main(String[] args)
 {
  int result, sum=0;
  int marks[] = {20,30,50};
  try{
      result = Integer.parseInt[args0]/
Integer.parseInt[args1] // line 1
      System.out.println(“Result is ‟ + result);
      for(int i=0;i<4;i++){
          sum += marks[i]; // line 2
      }
```

Handling Multiple Exceptions in a Single `catch` Block [3-3]

```
        System.out.println("Sum is " + sum);
    }catch(ArrayIndexOutOfBoundsException|ArithmeticExce
ption ex){
           // Catching multiple exceptions
               throw ex;
    }
  }
}
```

Using `try-with-resources` and `AutoCloseable` Interface [1-6]

- The `try-with-resources` statement is a `try` statement that declares one or more resources.
- A resource is an object that must be closed after the program is finished with it.
- The `try-with-resources` statement is written to ensure that each resource is closed at the end of the statement.
- Any object that implements `java.lang.AutoCloseable`, which includes all the objects which implement `java.io.Closeable`, can be used as a resource.
- The `AutoCloseable` interface is used to close a resource when it is no longer needed.

Using `try-with-resources` and `AutoCloseable` Interface [2-6]

- The `Closeable` interface extends the `AutoCloseable` interface.
- A `Closeable` is a source or destination of data that can be closed.
- The `close()` method is invoked to release resources that the object is holding, such as open files.
- The `close()` method of the `Closeable` interface throws exceptions of type `IOException` while the `close()` method of the `AutoCloseable` interface throws exceptions of type `Exception`.
- Consequently, subclasses of the `AutoCloseable` interface can override this behavior of the `close()` method to throw specialized exceptions, such as `IOException`, or no exception at all.

Using try-with-resources and AutoCloseable Interface [3-6]

The following Code Snippet explains the use of `try-with-resources`:

Code Snippet

```
public void writeToFile(String path){
 // Creating a try-with-resources statement
    try (Writer output = new BufferedWriter(new
FileWriter(path))){
        output.write("This is a sample statement.");
    } catch(IOException ex){
         System.out.println(ex.getMessage());
    }
}
```

Using try-with-resources and AutoCloseable Interface [4-6]

- Prior to Java SE 7, the `finally` block was used to ensure that a resource is closed regardless of whether the `try` statement completes normally or abruptly.
- The following Code Snippet shows an example that uses a `finally` block instead of a `try-with-resources` statement to close the resources:

Code Snippet

```
static void writeToFile(String path) throws IOException {
  Writer output = new BufferedWriter(new FileWriter(path));
  try {
    output.write(“This is a sample statement.”);
  } finally {
      if (output != null)
       output.close();
  }
}
```

Using `try-with-resources` and `AutoCloseable` Interface [5-6]

- Java SE 7 allows declaring one or more resources in a `try-with-resources` statement.
- The following Code Snippet shows an example that declares more than one resource in a single `try-with-resources` statement:

Code Snippet

```
public static void writeToFileContents(String 
sourceFile, String targetFile) throws 
java.io.IOException {

  // Declaring more than one resource in the try-with-
resources statement

     try (

       BufferedReader objBr = new BufferedReader(new 
FileReader(sourceFile));

       BufferedWriter output = new BufferedWriter(new 
FileWriter(targetFile))

     )
```

Using try-with-resources and AutoCloseable Interface [6-6]

```
        {
           // code to read from source and write to target
file.
        }
}
```

In this code:

- The `try-with-resources` statement contains two declarations that are separated by a semicolon: `BufferedReader` and `BufferedWriter`.
- When the block of code that directly follows the declaration, terminates either normally or due to an exception, the `close()` methods of the `BufferedWriter` and `BufferedReader` objects are automatically called in this order.
- That is, the `close()` methods of resources are called in the opposite order of their creation.

Enhancements In Exceptions in Java SE 7

- Multi-catch statements has helped the programmers to program more efficiently and concisely.
- Multi-catch statements also allow the programmer to handle a part of the exception and let it bubble up using the re-throw.
- `try-with-resources` statement facilitates less error-prone exception cleanup.

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 [1-2]

- To create a user-defined exception class, the class must inherit from the `Exception` class.
- The syntax is as follows:

Syntax

```
public class <ExceptionName> extends Exception {}
```

- The following Code Snippet explains creation of a user-defined exception class:

Code Snippet

```
// Creating a user-defined exception class
public class ServerException extends Exception{
 public ServerException()
  {}
 // Overriding the getMessage() method
    @Override
```

Creating a User-defined Exception [2-2]

```
     public String getMessage() // line 1
     {
       return “Connection Failed”;
     }
}
```

In the code:

- ServerException is a user-defined exception class that inherits from the built-in Exception class.
- The getMessage() method of the Exception class has been overridden in the ServerException class to print a user-defined message “Connection Failed”.

Throwing User-defined Exceptions [1-2]

- To raise a user-defined exception, a method must throw the exception at runtime.
- The exception is transferred further up in the call stack and handled by the caller of the method.
- The following Code Snippet explains how to throw a user-defined exception:

Code Snippet

```
// creating a class to use the user-defined exception
class MyConnection {
 String ip;
 String port;
 public MyConnection()
 {}
```

Throwing User-defined Exceptions [2-2]

```
public MyConnection(String ip, String port) {
  this.ip=ip;
  this.port=port;
}
// creating a method that throws the user-defined
exception
public void connectToServer() throws ServerException{
// line 1
 if(ip.equals("127.10.10.1") && port.equals("1234"))
  System.out.println("Connecting to Server…");
 else
  throw new ServerException(); // line 2 – throwing
the exception
 }
}
```

Wrapper Exceptions [1-5]

- Exception wrapping is catching an exception, wrapping it in a different exception, and throwing the wrapper exception.
- Exception wrapping is a standard feature in Java since JDK 1.4.
- Most of Java's built-in exceptions have constructors that can take a 'cause' parameter.
- They also provide a `getCause()` method that will return the wrapped exception.
- The main reason for using exception wrapping is to prevent the code further up the call stack from knowing about every possible exception in the system.
- Also, one may not want the top level components to know anything about the bottom level components and the exceptions they throw.

Wrapper Exceptions [2-5]

The following Code Snippet explains the use of wrapper exceptions:

Code Snippet

```
// creating a user-defined exception class
class CalculatorException extends Exception{ // line 1
 public CalculatorException()
  {}
// constructor with Throwable object as parameter
 public CalculatorException(Throwable cause){
  super(cause);
 }
 // constructor with a message string and Throwable
object as parameter
 public CalculatorException(String message, Throwable
cause){
  super(message, cause);
 }
```

Wrapper Exceptions [3-5]

```
// creating the Calculator class
class Calculator { // line 2
 // method to divide two numbers
 public void divide(int a, int b) throws
CalculatorException // line 3
 {
  // try-catch block
  try{
   int result = a/b; // performing division
   System.out.println("Result is " + result);
  }
  catch(ArithmeticException ex)
  {
   // throwing the wrapper exception – line 4
throw new CalculatorException("Denominator cannot be
zero", ex);
```

Wrapper Exceptions [4-5]

```
    }
   }
}
// creating the TestCalculator class
public class TestCalculator {
public static void main(String[] args){
  try{
   // creating object of Calculator class
   Calculator objCalc = new Calculator();
   // invoking the divide method
   objCalc.divide(10,0);
  }catch(CalculatorException ex){
    // getting the cause from the wrapper
    Throwable t = ex.getCause(); // line 5
    // printing the message and the cause
```

Wrapper Exceptions [5-5]

```
    System.out.println("Error: "+ ex.getMessage()); //
line 6
     System.out.println("Cause: " + t); // line 7
    }
   }
}
```

Assertions [1-6]

- An assertion is 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 it is not true, 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 [2-6]

- 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 [3-6]

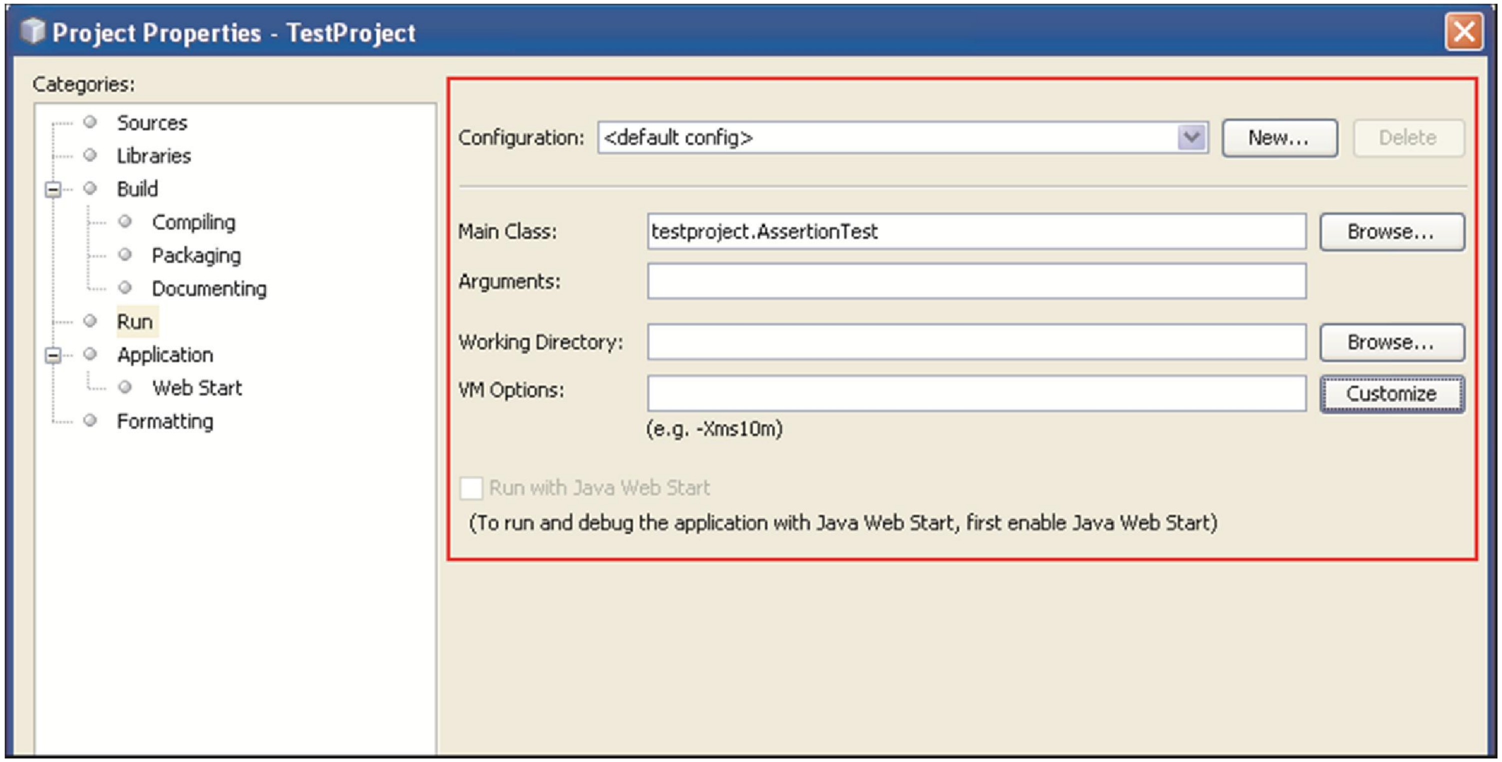
- To ensure that assertions do not become a performance liability in deployed applications, assertions can be enabled or disabled when the program is started.
- Assertions are disabled by default.
- Disabling assertions removes their performance related issues entirely.
- Once disabled, they become empty 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 [4-6]

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:

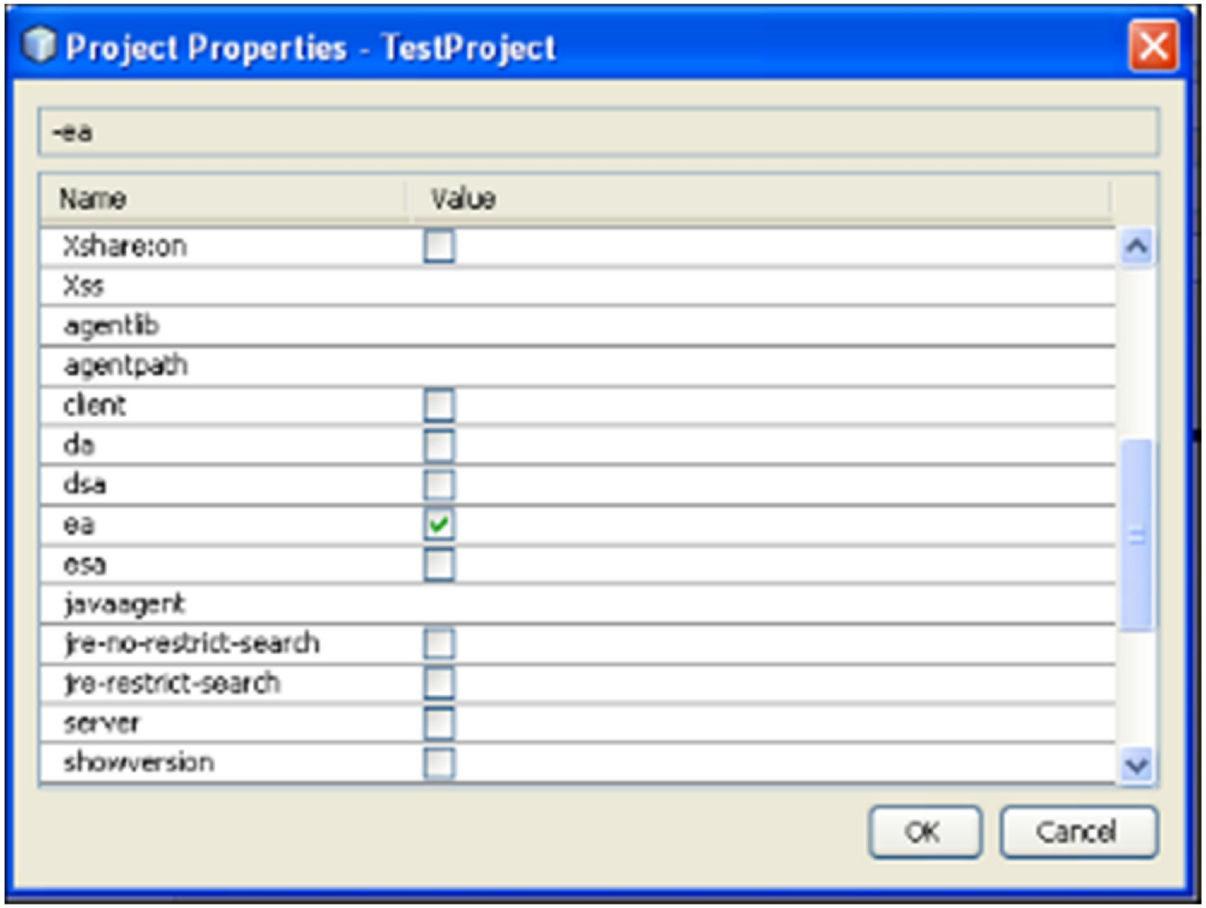
Working Directory:

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 [5-6]

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

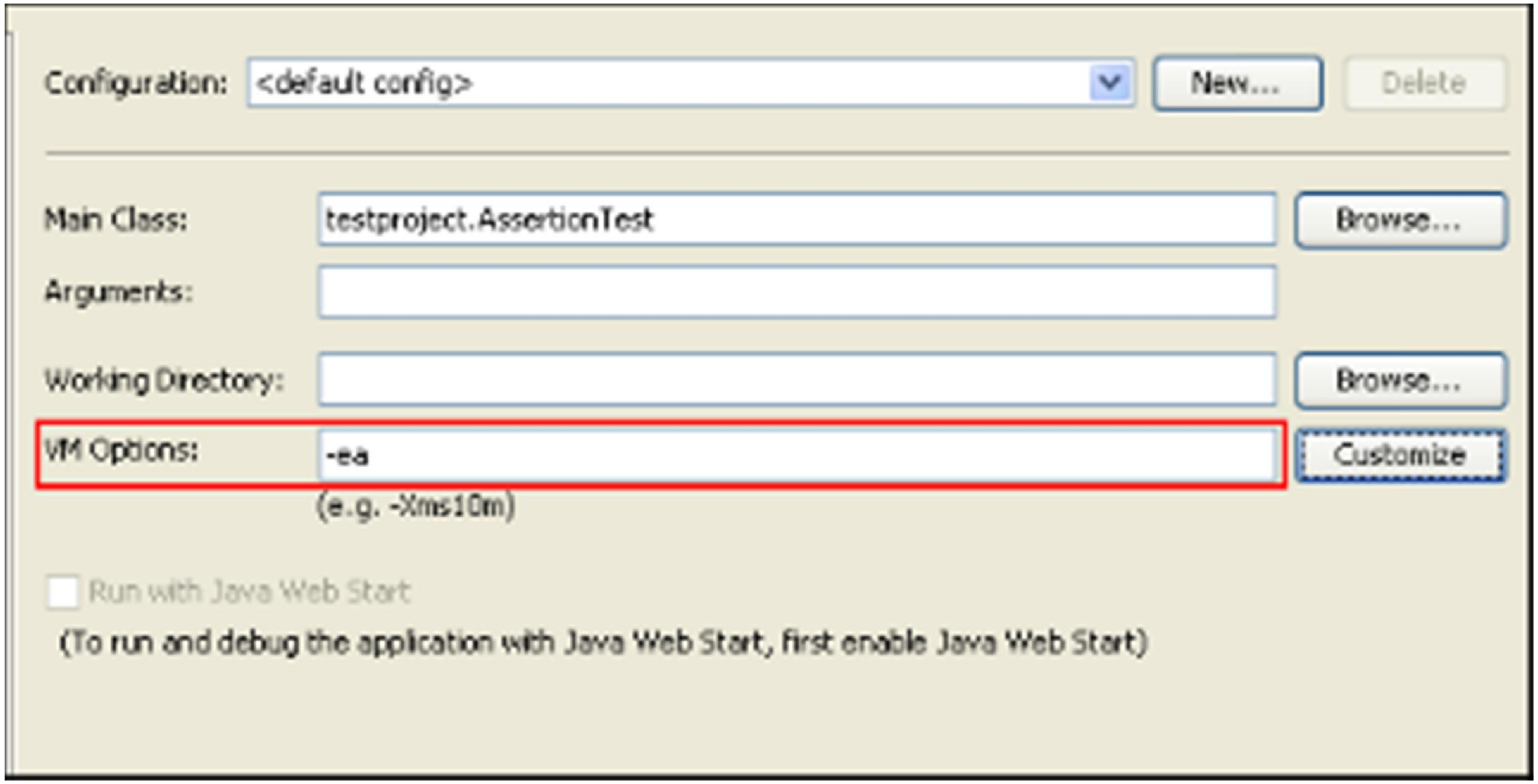


-ea

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

Assertions [6-6]

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**.

Internal Invariants [1-3]

- Earlier, where assertions were not available, many programmers used comments to indicate their assumptions concerning a program's behavior.
- For example, one might have written a comment as shown in the following Code Snippet to explain the assumption about an `else` clause in an `if...else` statement.

Code Snippet

```
public class AssertionTest{
public static void main(String[] args){
  int a = 0;
if(a>0)
   // do this if a is greater than zero
   else{
    // do that, unless a is negative
    }
  }
}
```

Internal Invariants [2-3]

- The code states that the `else` statement should be executed only if `a` is equal to zero but not if `a>0`.
- However, at runtime, this can be missed out since no error will be raised even if a negative number is specified at runtime.
- For such invariants, one can use assertion as shown in the following Code Snippet:

Code Snippet

```
public class AssertionTest{
public static void main(String[] args){
  int a = -1;
  if(a>0)
   System.out.println(“Greater than zero”);
else{
```

Internal Invariants [3-3]

```
        assert a==0:"Number should not be negative";
        System.out.println("Number is zero");
      }
    }
  }
```

In the code:

- The value of `a` has been set to `-1`.
- In the `else` block, an `assert` statement is provided which checks if `a` is equal to zero.
- If not, the detail message will be displayed to the user and the application will terminate.

Control Flow Invariants [1-3]

- Assertions can also be applied to control flow invariants such as a `switch` statement that has no default case.
- The absence of a default case is indicative of the belief that one of the cases will always be executed.
- The assumption that a particular variable will surely have any one of a small set of values is an invariant that needs to be checked with an assertion.

Control Flow Invariants [2-3]

Suppose a switch statement appears in a program that checks days of a week as shown in the following Code Snippet:

Code Snippet

```
public class ControlFlowTest {
 public static void main(String[] args) {
  String day=args[0];
  switch (day) {
   case “Sun”:
    // do this
    break;
   case “Mon”:
   // do this
   break;
   case “Tue”:
```

Control Flow Invariants [3-3]

```
      // do this
      break;
   case "Wed":
   // do this
   break;
   case "Thu":
   // do this
   break;
   case "Fri":
   // do this
   break;
   case "Sat":
   // do this
   break;
  }
 }
}
```

PreCondition, PostCondition, and Class Invariants [1-7]

While the assert construct is not a complete help in itself, it can help support an informal design-by-contract style of programming. One can use assertions for:

- **Preconditions**: what must be true when a method is invoked?
- **Postconditions**: what must be true after a method executes successfully?
- **Class invariants**: what must be true about each instance of a class?

PreCondition, PostCondition, and Class Invariants [2-7]

Preconditions:

- By convention, preconditions on public methods are enforced by making explicit condition checks that throw particular, specified exceptions.
- Consider the code given in the following Code Snippet:

Code Snippet

```
// method to set the refresh rate and
//throw IllegalArgumentException if rate <=0 or rate >
//MAX_RATE
public void setRate(int rate) {
  // Apply the specified precondition in public method
  if (rate <= 0 || rate > MAX_REFRESH_RATE)
    throw new IllegalArgumentException("Illegal rate: " +
rate);
  setInterval(1000/rate);
}
```

PreCondition, PostCondition, and Class Invariants [3-7]

- One can use an assertion to test a precondition of a non-public method that is believed to be true no matter what a user does with the class.
- An assertion is appropriate in the helper method `setInterval(int interval)` that is invoked by the `setRate()` method as shown in the following Code Snippet:

Code Snippet

```
// Method to set the refresh interval (in
milliseconds) which
// must correspond to a legal frame rate
  private void setInterval(int interval) {
   // Verify the adherence to precondition in the non-
public method
   assert interval > 0 && interval <= 1000/MAX_RATE :
interval;
    // Set the refresh interval
  System.out.println("Interval is set to:" +
interval);
  }
```

PreCondition, PostCondition, and Class Invariants [4-7]

Postconditions:

- Postconditions can be checked with assertions in both public and non-public methods.
- The public method **`pop()`** in the following Code Snippet that uses an `assert` statement to check a postcondition:

Code Snippet

```
public class PostconditionTest{
ArrayList values = new ArrayList();
 public PostconditionTest(){
  values.add("one");
  values.add("two");
  values.add("three");
  values.add("four");
 }
```

PreCondition, PostCondition, and Class Invariants [5-7]

```
 public Object pop(){
  int size = values.size(); // line 1
  if(size == 0)
   throw new RuntimeException(“List is empty!!”);
   Object result = values.remove(0) ;
// verify the postcondition
  assert(values.size() == size -1); // line 2
 return result;
 }
}
```

PreCondition, PostCondition, and Class Invariants [6-7]

Class Invariants:

- A class invariant is a type of internal invariant that is applied to every instance of a class.
- It is applicable at all times except when the instance is transiting from one consistent state to another.
- A class invariant can be used to specify the relationships among multiple attributes.
- Also, it should be true before and after any method completes.
- The assertion mechanism does not adopt any specific style for checking invariants.
- However, it is sometimes convenient and advisable to combine the expressions that verify the required constraints into a single internal method that can be called by assertions.

PreCondition, PostCondition, and Class Invariants [7-7]

- With respect to the balanced tree example, it would be better to implement a private method that checked that the tree was indeed balanced as per the rules of the data structure as shown in the following Code Snippet:

Code Snippet

```
// Returns true if this tree is properly balanced
 private boolean balanced() {
  // code to check if the tree is balanced
  }
```

- Since this method is used to check a constraint that should be true before and after any method completes, each `public` method and constructor should contain the line, `assert balanced();` immediately prior to its return.

Summary [1-2]

- An exception is an event occurring during program execution that leads to disruption of the normal flow of the program's instructions.
- The finally block is executed even if an exception occurs in the try block.
- The throws clause is written after the method name and argument list and before the opening brace of the method.
- All methods use the throw statement to throw an exception.
- The throw statement takes a single argument which is a throwable object.
- To create a multiple exception catch block, the types of exceptions that catch block can handle are specified separated by a vertical bar (|).
- The try-with-resources statement is a try statement that declares one or more resources.
- A resource is an object that must be closed after the program is finished with it.

Summary [2-2]

- To create a user-defined exception class, the class must inherit from the Exception class.
- Exception wrapping is catching an exception, wrapping it in a different exception, and throwing the wrapper exception.
- 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.