Power Programming with Java

Session: 1

Exceptions and Assertions

For Aptech Centers Use Only

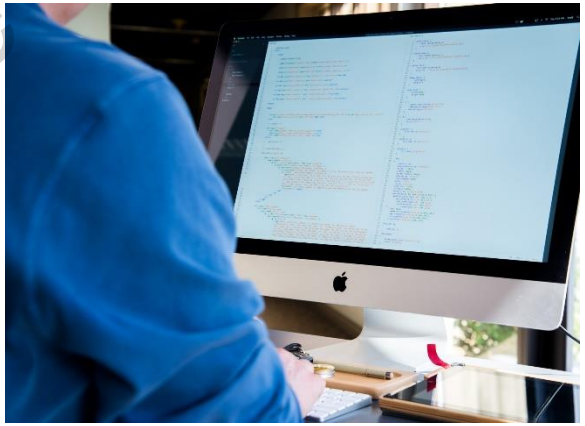
Objectives

- 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

For Aptech Centre Use Only

Introduction

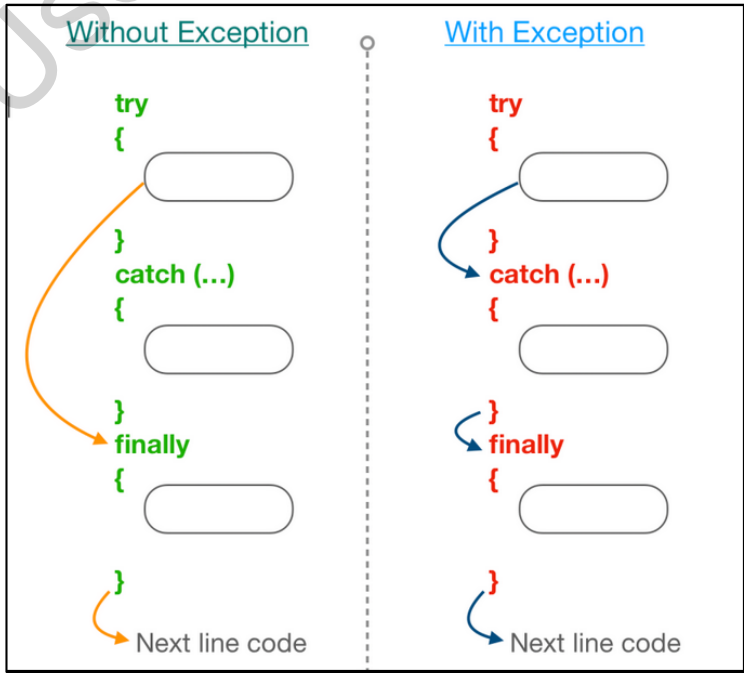
- Java programming language provides the ability to handle unexpected events in the program.
- This is because, even though a code is well written, it is prone to behave erroneously.
- With the help of exception handling, the developer can ensure that the user gets a proper message in case some error 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.



For Aptech Centre Use Only

Overview of Exceptions

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

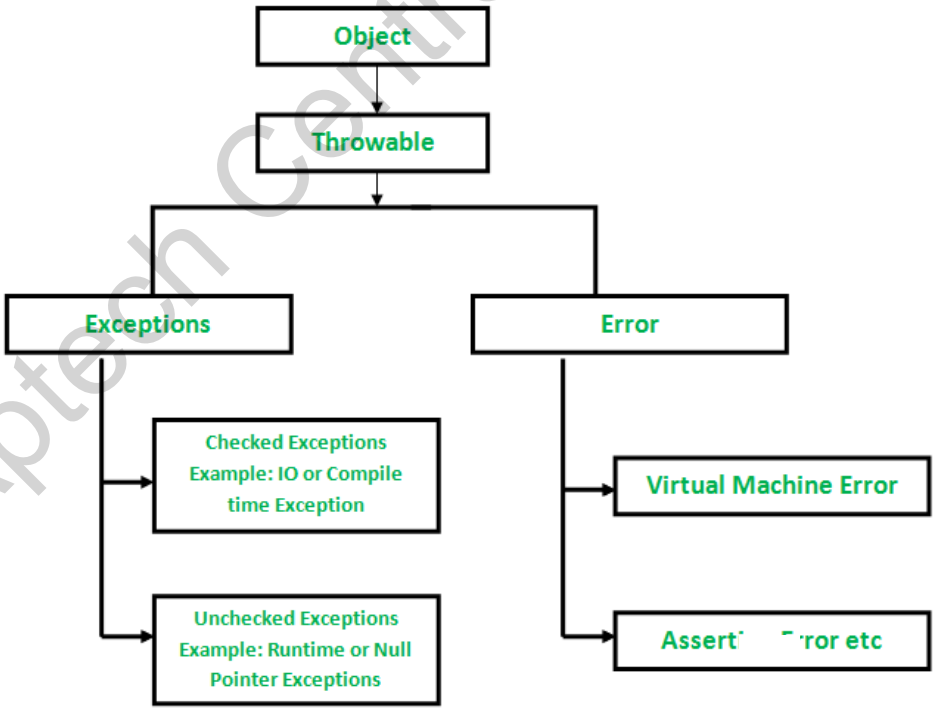


For Aptech Centre Use Only

Exception Handling [1-2]

- Exceptions and error types are sub classes of `Throwable` class and are used to handle exceptions that occur during execution of a program.
- For example,

`NullPointerException`, `IndexOutofBoundsException` and so on fall under Exception.



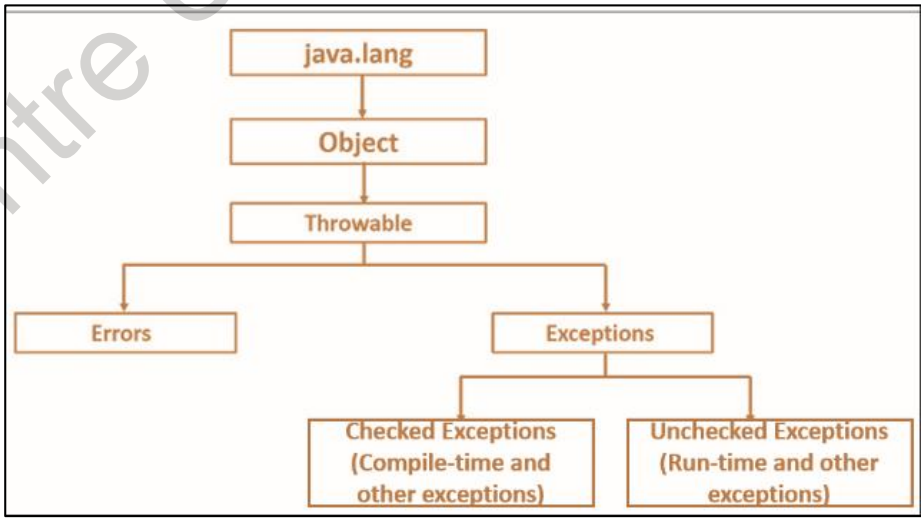
Class Hierarchy for Exceptions and Errors

For Aptech Centre Use Only

Exception Handling [2-2]

Oracle classifies exceptions under three categories as follows:

- Checked Exception
- Unchecked Exception
- Error



- An object is created by a method if an exception occurs within a method.
- Creating an `Exception` object and handing it to the runtime system is known as throwing an exception.
- Five keywords are used to handle Java exceptions. They are namely, `try`, `catch`, `throw`, `throws`, and `finally`.

For Aptech Centre Use Only

try-catch-finally Block [1-4]

- To handle exceptions, developer must identify 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.

Syntax

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

For Aptech Centre Use Only

try-catch-finally Block [2-4]

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());
       }
   }
}
```

For Aptech Centre Use Only

try-catch-finally Block [3-4]

- The `finally` block is executed even if an exception occurs in the `try` block.
- It helps 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 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.

For Aptech Centre Use Only

try-catch-finally Block [4-4]

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

For Aptech Centre Use Only

throw and throws Keywords [1-5]

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.

For Aptech Centre Use Only

throw and throws Keywords [2-5]

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");
       }
  }
}
```

For Aptech Centre Use Only

throw and throws Keywords [3-5]

Java platform provides several exception classes which are descendants of `Throwable` class.

These classes allow programs to differentiate among different types of exceptions that can occur during execution of a program.

All methods use 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.

For Aptech Centre Use Only

throw and throws Keywords [4-5]

Code Snippet shows use of `writeToFile()` method to throw `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 {
              if (objPwOut != null) {
                  objPwOut.close();
                  System.out.println("PrintWriter closed"); }
        }
    }
```

For Aptech Centre Use Only

throw and throws Keywords [5-5]

```
        public static void main(String[] args)
        {
         try{
               FileWriting fw = new FileWriting();
               fw.writeToFile();
         } catch(FileNotFoundException ex)
            {
               // 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.

For Aptech Centre Use Only

Throwing Exceptions from Methods

One can directly throw exceptions from a method and transfer it to a method higher up in the hierarchy. Consider 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);
 }
}
public class TestCalculator {
 public static void main(String[] args) {
   try{
        Calculator objCalc = new Calculator();
        objCalc.divide(Integer.parseInt(args[0]), Integer. parseInt(args[1]));
   }catch(ArithmeticException ex){
       System.out.println("Denominator cannot be set to zero");
   }
  }
}
```

For Aptech Centre Use Only

Handling Multiple Exceptions in a Single catch Block [1-2]

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

Syntax

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

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

For Aptech Centre Use Only

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

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

For Aptech Centre Use Only

Using try-with-resources and AutoCloseable Interface [1-5]

- `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.
- `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 required.

For Aptech Centre Use Only

Using try-with-resources and AutoCloseable Interface [2-5]

- `Closeable` interface extends `AutoCloseable` interface.
- A `Closeable` is a source or destination of data that can be closed.
- `close()` method is invoked to release resources that the object is holding, such as open files.
- `close()` method of `Closeable` interface throws exceptions of type `IOException` while `close()` method of `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.

For Aptech Centre Use Only

Using try-with-resources and AutoCloseable Interface [3-5]

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());
    }
}
```

For Aptech Centre Use Only

Using try-with-resources and AutoCloseable Interface [4-5]

- Prior to Java SE 7, `finally` block was used to ensure that a resource is closed, regardless of whether `try` statement completes normally or abruptly.
- 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();
  }
}
```

For Aptech Centre Use Only

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

- Java SE 7 and later allow declaring one or more resources in `try-with-resources` statement.
- Following Code Snippet 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))
       )
       {
          // 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.

For Aptech Centre Use Only

Other Features of Exception Handling

More features of exception handling are as follows:

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

For Aptech Centre Use Only

The java.lang.IllegalCallerException Class

- To overcome existing issues that arose when trying to get stack trace, the Stack-Walking API was introduced in Java 9.
- This API comprises `java.lang.StackWalker` class, along with a nested class `Option` and `StackFrame` interface.
- The Stack-Walking API also includes the `java.lang.IllegalCallerException` class.
- This class represents an exception that can be thrown to indicate that a method has been called by an inappropriate caller.

For Aptech Centre Use Only

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.

For Aptech Centre Use Only

Creating a User-defined Exception

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

Syntax

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

- 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
       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".

For Aptech Centre Use Only

Throwing User-defined Exceptions

- 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.
- 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()
  {}
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
 }
}
```

For Aptech Centre Use Only

Wrapper Exceptions [1-3]

- 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.

For Aptech Centre Use Only

Wrapper Exceptions [2-3]

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);
 }
// 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);
  }
```

For Aptech Centre Use Only

Wrapper Exceptions [3-3]

```
   catch(ArithmeticException ex)
   {
    // throwing the wrapper exception – line 4
   throw new CalculatorException("Denominator cannot be zero", ex);
   }
  }
}
// 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
    System.out.println("Error: "+ ex.getMessage()); // line 6
      System.out.println("Cause: " + t); // line 7
     }
    }
}
```

For Aptech Centre Use Only

Assertions [1-3]

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.

For Aptech Centre Use Only

Assertions [2-3]

- 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.

For Aptech Centre Use Only

Assertions [3-3]

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>`

For Aptech Centre Use Only

Internal Invariants [1-2]

- 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
   }
  }
}
```

For Aptech Centre Use Only

Internal Invariants [2-2]

- 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 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{
        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.

For Aptech Centre Use Only

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.

For Aptech Centre Use Only

Control Flow Invariants [2-3]

Suppose a `switch` statement appears in a program that checks days of a week as shown in 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":
      // do this
      break;
   case "Wed":
   // do this
   break;
   case "Thu":
   // do this
   break;
```

For Aptech Centre Use Only

Control Flow Invariants [3-3]

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

The code probably indicates an assumption that the **day** variable will have only one of the seven values. To test this assumption, one can add the following default case to the code:

```
default:
assert false : day + " is incorrect";
```

If the **day** variable takes on any other value and assertions are enabled, the assertion will fail and an AssertionError will be thrown.

For Aptech Centre Use Only

PreCondition, PostCondition, and Class Invariants [1-5]

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?

For Aptech Centre Use Only

PreCondition, PostCondition, and Class Invariants [2-5]

Preconditions:

- By convention, preconditions on public methods are enforced by making explicit condition checks that throw particular, specified exceptions.

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);
}
```

- 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 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);
   }
```

For Aptech Centre Use Only

PreCondition, PostCondition, and Class Invariants [3-5]

Postconditions:

- Postconditions can be checked with assertions in both public and non-public methods.
- The public method **`pop()`** in 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");
 }
 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;
 }
}
```

For Aptech Centre Use Only

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.

For Aptech Centre Use Only

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

- With respect to the balanced tree example, it would be better to implement a private method which checks that the tree was indeed balanced as per the rules of the data structure.
- This is shown in 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.

For Aptech Centre Use Only

Summary

- An exception is an event occurring during program execution that leads to disruption of the normal flow of the program's instructions.
- `try-catch` blocks are used to handle exceptions.
- 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.
- 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.

For Aptech Centre Use Only