Exception Handling

Unit 6

Pradeep LN

Objectives

- Error Handling in Conventional Languages
- Exceptions
- Exception Handling in Java
  - Exception Hierarchy
  - Throwable Class
  - Error and Exception
  - Types of Exceptions
  - Throwing Exceptions
  - Declaring Exceptions per Method
  - Catching Exception
  - Designing user-defined Exception classes
- Method Overriding and Exceptions
- Assertions
- Why log?
- Java SE Logging API
- Severity Levels
- Log Hierarchies

Error handling in traditional languages

- Traditional error handling methods include
  - Boolean functions (which return TRUE/FALSE).
  - Integer functions (returns –1 on error).
  - And other return arguments and special values.

```
int main () {
        int res;
        if (can_fail () == -1) {
                cout << "Something failed!" << endl;
                return 1;
        }
        if(div(10,0,res) == -1) {
                cout << "Division by Zero!" << endl;
               return 2;
        }
              return 0;
  }
```

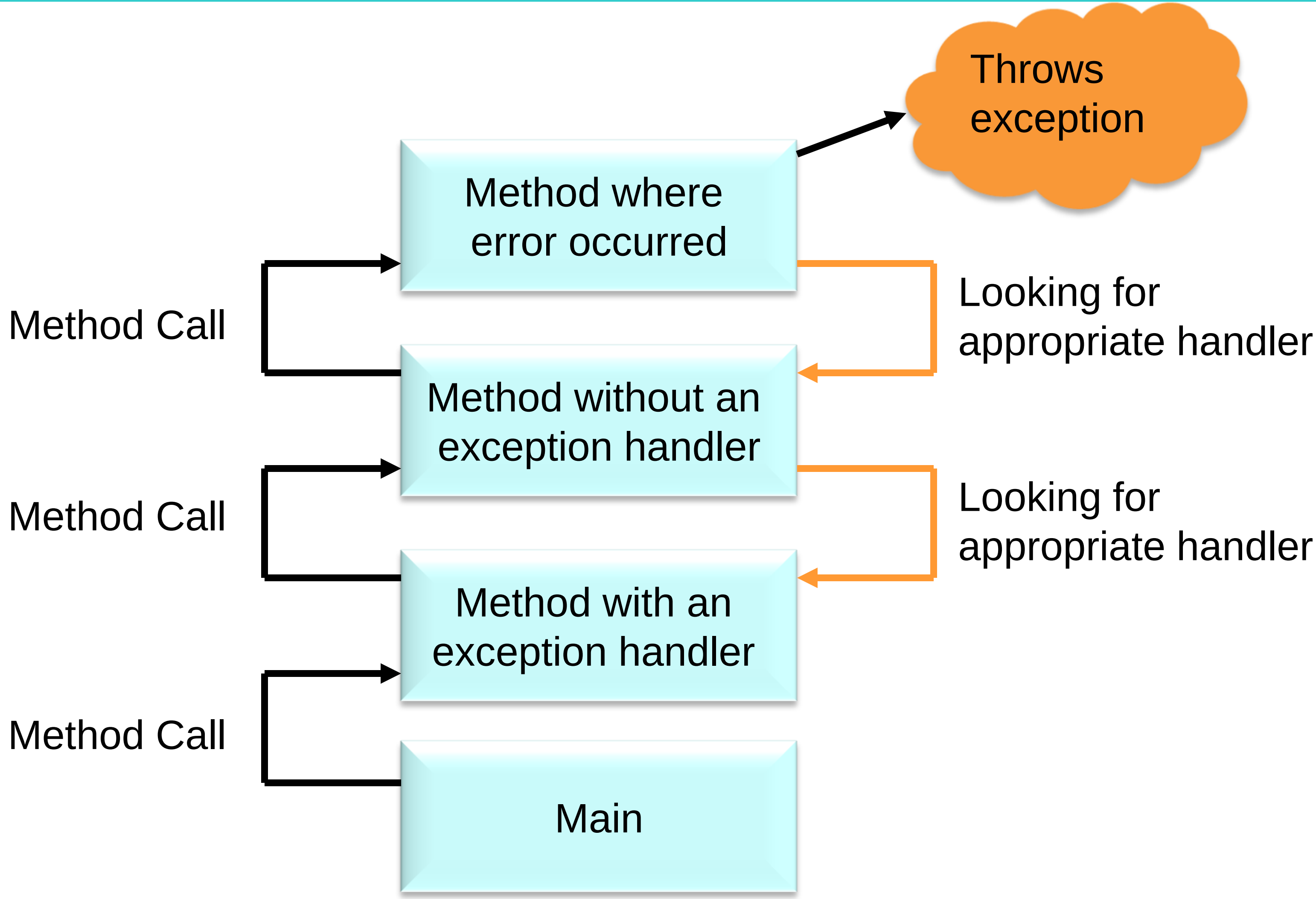
Problems associated with Traditional approach

- The caller of the function is notified if the function was executed successfully or not by returning a value.
- With this approach, a function can only return a value to indicate if the function executed successfully or not and cannot return any other value.
- Since the function returns a value, the caller is not bound or forced to receive the same and can simply ignore it.
- Any error handling will be done in a series of if….else blocks, which leads to cluttered code.
- With no clear separation of error handling code, maintainability of the code becomes tedious.

Error handling in Java

- Java’s support for error handling is done through Exceptions.
- **What is an Exception?**
    - An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
    - When an error occurs within a method, the caller of the method has to be notified about the error, here the method creates an *exception object* and hands it off to the caller of the method.
    - The exception object, contains information about the error, including its type and state of the program when the error occurred.
    - Creating an exception object and notifying the caller of the method is called *throwing an exception*.

Exception handling mechanism

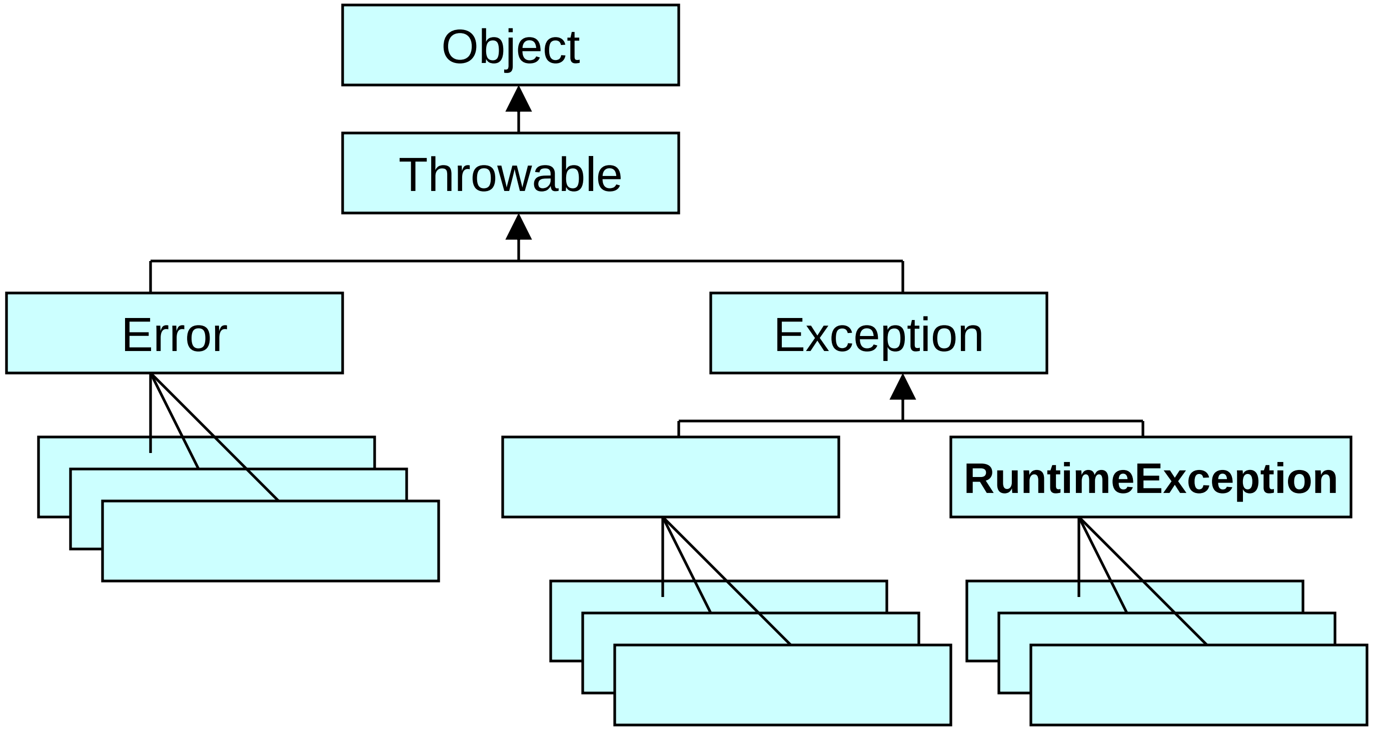

Throws exception
Method where error occurred
Method Call
Method without an exception handler
Looking for appropriate handler
Method Call
Method with an exception handler
Looking for appropriate handler
Method Call
Main

Why use exceptions?

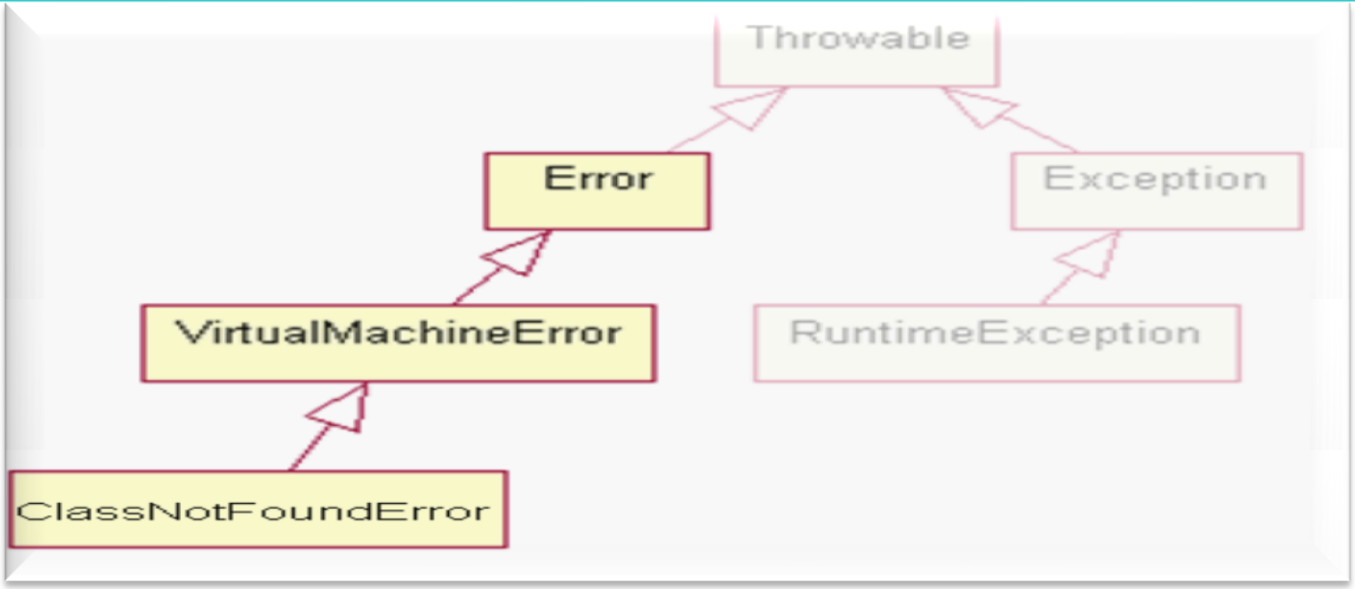
- Exception handling provides the programmer with many advantages over traditional error-handling techniques:
  - It provides a means to separate error-handling code from functioning program code.
  - It provides a mechanism for propagating errors up the method call stack, meaning that methods higher up the call chain can be allowed to handle problems originating lower in the chain.
  - It provides a way to organize and differentiate between different types of abnormal conditions.
  - Both, application logic and problem handling logic, become simpler, cleaner and clearer.

Throwable class

- When an error occurs in a method the caller of the method is notified by throwing an exception object, this object should be of the type class ***Throwable***.
- *Throwable* has two subclasses, *Error* and *Exception*.



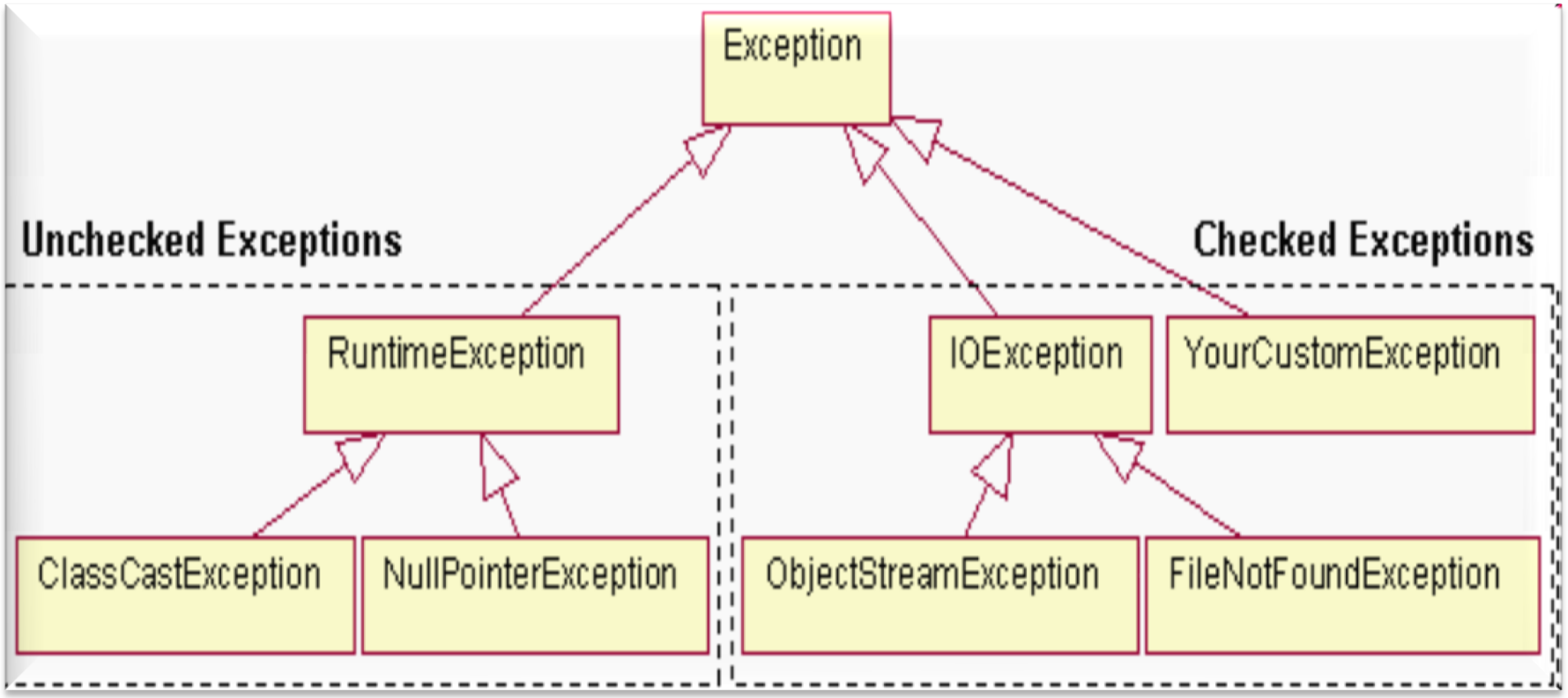
Errors



- Errors are exceptional conditions that are external to the application, and that the application cannot anticipate or recover from, therefore Errors are not handled by the application.
- For example,
  - An application has to open a file and write to it, due to some hardware error, the application is unable to open the file we would get a java.io.IOError.

Types of Exception

- Exceptions are of 2 types
  - **Checked exceptions**
  - **Unchecked or runtime exception**



Types of Exception

- **Checked exceptions**
  - These are exceptional conditions that a well written application should anticipate and recover from.
  - Since these exceptions are anticipated and are recoverable conditions, compiler forces these exceptions to be handled.

  e.g.: FileNotFoundException
- **Unchecked or runtime exception**
  - These are exceptional conditions that are internal to the application, and that the application cannot anticipate or recover from.
  - These usually indicate programming bugs, such as logical errors or improper use of an API.
  - Since these can be fixed programmatically, compiler does not force these exceptions to be handled.

  e.g.: NullPointerException

Throwing an Exception

- All methods use the ***throw*** statement to throw an exception.
- The ***throw*** statement requires a throwable object as argument. A throwable object are instances of any subclass of the ***Throwable*** class.
- ***throw*** causes the method to terminate and returns an exception object to the caller.

```
public Object pop()
{
        if(size == 0)
                throw new EmptyStackException();
        return objectAt(size--);
}
```

Exception Handlers

- Java programming language supports, exception handling by providing three exception handler components, these are
- ***try***
- ***catch***
- ***finally***

- The *try*, *catch* and *finally* blocks are used to write an exception handler.

try Block

- Any code that might throw an exception is enclosed within a ***try*** block. It is the first step in constructing an exception handler.
- A try block must be followed by at least by ***one catch block or one finally block***.

```
try
{
        code
                                        try block
        ………..
}
catch and finally blocks……
```

try Block Example

```
public int countChars(String fileName) {
        int total = 0;
        try {
                FileReader r = new FileReader(fileName);
                while( r.ready()) {
                        r.read();
                        total++;
                }
                r.close();
            }
            catch……
}
```

catch Block

- A ***catch*** block is an exception handler associated with a ***try*** block and handles the type of exception indicated by its argument.
- The argument type, indicates the type of exception that the handler can handle and must be the name of a class that inherits the Throwable class.
- The ***catch*** block contains code that is executed if and when the exception handler is invoked.
- Every try block is associated with ***zero or more catch block*** and no code can be between the end of try block and beginning of the first catch block.

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

catch Block Example

```
public int countChars(String fileName) {
        int total = 0;
        try {
                FileReader r = new FileReader(fileName);
                while( r.ready()) {
                        r.read();
                       total++;
               }
                       r.close();
        }
        catch(FileNotFoundException e){
              System.out.println("File named " + fileName + "not found. " +e);
                total = -1;
        }
        catch(IOException e){
           System.out.println("IOException occured "+ "while counting " +e);
               total = -1;
        }
}
```

See listing : **ExceptionDemo1.java**

finally Block

- A ***finally*** block is executed irrespective of whether the try block throws an error or not.
- ***finally*** block is guaranteed to be executed and can be used for any clean-up code.
- If no catch block matches an exception thrown, control is transferred to the ***finally*** block.
- There can be only one ***finally*** block for a try block.

```
try{

}

catch(ExceptionType name){

}

finally{

}
```

finally Block Example

```
public int countChars(String fileName) {
        int total = 0;
        FileReader r = new FileReader(fileName);
        try {
                while( r.ready()) {
                        r.read();
                         total++;
                }
        }
        catch(FileNotFoundException e){
                System.out.println("File named " + fileName + "not found. " + e);
                total = -1;
        }
        catch(IOException e){
              System.out.println("IOException occured "+ "while counting " + e);
               total = -1;
        }
        finally {
                r.close();
         }
}
```

Clean up code goes into finally block

See listing : ExceptionDemo2.java

StackTrace

```
public static void main(String[ ] args) {
        try {
                method1();
                method2();
                method3();
                ……………
                ……………

        }
        catch(Exception e) {
                e.printStackTrace();
        }
}
```

try block has several method calls which can throw an exception

Use the **printStackTrace()** method, to trace the origin of the Exception

See listing : **StackTraceDemo.java**

Specifying a method as throwing Exception

```
public void writeList() {
        PrintWriter out = new PrintWrite
                                          FileWrite
        for(int i=0 ; i<SIZE ; ++i)
                out.println("Value at" + vector.elementAt(i));
        out.close();
}
```

writeList() method can throw an IOException or ArrayIndexOutOfBounds Exception

- The writeList() method can anticipate an exception and catch the same.
- The method however need not catch the exception and thereby allow a method further up the call stack to handle it.
- In that scenario the method has to specify the exception being thrown by the method.

Specifying a method as throwing Exception

- A method can specify to throw an exception by adding a ***throws*** clause to the method declaration.
- The throws clause comprises the ***throws*** keyword followed by a comma-separated list of all the exceptions thrown by that method.
- The clause goes after the method name and argument list and before the brace that defines the scope of the method.

```
public void writeList() throws IOException ,
                                  ArrayIndexOutOfBoundsException  {

}
```

- **Here ArrayIndexOutOfBoundsException is an unchecked exception, including it in the throws clause is not mandatory.**

User defined Exceptions

- Java platform provides a lot of exception classes for use, else the programmer can write his own exception class.
- If an exception cannot be represented by those in the Java platform, a user can define his own exception.
- This exception class should be a subclass of the Exception class or any of its subclass.

```
public class
ProductNotFoundException
extends Exception

{

}
```

```
public Product getProducts()
throws
ProductNotFoundException {

        ………

        ………

}
```

Method overriding and Exceptions

- The checked exception classes named in the throws clause are part of the contract between the implementer and user of the method or constructor.
- The throws clause of overriding method can specify all or none of the exception classes specified in the throws clause of the overridden method in the super class.
- The overriding method may not specify that this method will result in throwing any checked exception which the overridden method is not permitted.
- However the overriding method can throw exceptions which are subclasses of the exceptions in the throws clause of the overridden method.

Exercise

- Create a class with a main( ) that throws an object of class Exception inside a try block. Give the constructor for Exception a String argument. Catch the exception inside a catch clause and print the String argument. Add a finally clause and print a message to prove you were there.
- See ExceptionExercise.doc

What is an Assertion?

- An *assertion* is a statement in Java that enables you to test the assumptions about your program.
- Assertions are one of the quickest and most effective ways to detect and correct bugs at development time.
- Each assertion contains a boolean expression that you believe will be true when the assertion executes. If its 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, thereby validating the program to be free of errors.

```
public class MyClass {
    public void aMethod() {
        Employee emp = null;
        //Get a Employee object
//Check to ensure we have one
        assert emp != null;
    }
}
```

One of the most important features of this facility is that these checks can be turned on and off at runtime.

Simple Assertion Form

- The assertion statement has two forms
- The first is:

  assert *Expression1* ;

- Where *Expression1* is a boolean expression
- When the system runs the assertion, it evaluates *Expression1* and if it is false throws an **AssertionError** with no detail message.

```
public void display(String empId) {
        Employee emp = getEmployee(empId);
        assert emp != null;
        System.out.println("Employee name "+emp.getName());
        System.out.println("Employee address "+emp.getAddress());
}
```

Complex Assertion Form

- The second form of the assertion statement is:

  assert *Expression1* : *Expression2* ;

- where:
  - *Expression1* is a boolean expression
  - *Expression2* is an expression that has a value
  - *Expression2* cannot be invocation of a method that returns **void.** It can be an object, boolean, char, int, long, float or double.

```
public void display(String empId) {
        Employee emp = getEmployee(empId);
        assert emp != null : “Employee object not found “;
        System.out.println(“Employee name “+emp.getName());
        System.out.println(“Employee address “+emp.getAddress());
}
```

Compiling files that use Assertions

- In order for the javac compiler to accept code containing assertions, use the –source 1.4 command-line option.

- Example

  javac –source 1.4 MyClass.java

- Assertions were introduced from J2SE 1.4, so the flag is a must and necessary so as to not cause source compatibility problems.

See listing : **AssertionDemo.java**

Enabling and Disabling Assertions

- By default, assertions are disabled at runtime. Two command line switches allow to selectively enable or disable assertions.
- To enable assertions at various granularities, use the

  -enableassertions, or **–ea** , switch
- To disable assertions at various granularities, use the

  -disableassertions, or **-da**, switch
- Specify the granularity with the arguments that you provide to the switch

Arguments

- *no arguments*

  Enables or disables assertions in all classes except system classes

- *packageName...*

  Enables or disables assertions in the named package and any subpackages

- ...

  Enables or disables assertions in the unnamed package in the current working directory

- *className*

  Enables or disables assertions in the named class

Where to use Assertions

- The assert construct is not a full-blown design-by-contract facility, it can help support an informal design-by-contract style of programming.
- Assertions can be used for –
  - *Preconditions* – what must be true when a method is invoked.
  - *Postconditions* – what must be true after a method completes successfully.
  - *Class invariants* – what must be true about each instance of a class.
- We will look at each of the conditions with an example of a integer stack which provides operations such as push to add an item and pop to retrieve an item from stack.

Preconditions

- To withdraw amount from an account the presented amount should not be negative. This precondition can be programmed using assertions as follows

```
public void withdraw(int amount)
 {
        //precondition
        assert amount >0 : “Amount negative“;
        …….
}
```

Postconditions

- After successful withdrawal the balance should have reduced. This post condition can be programmed using assertions as follows

```
public void withdraw(int amount) {
        int oldBalance = balance;
        if(balance > amount)
                balance -= amount;
        assert  balance < oldBalance : “balance never changed“;
     …….
}
```

Class Invariants

- A class invariant is a condition that must be true before and after any method completes.
- In the Account example, at any point of time the balance should be greater than mini balance. These conditions, can be coded as follows.

```
public void withdraw(int amount) {
        ........
        assert  balance > miniBalance : “balance less than min balance“;
        .......
}
```

Where not to use Assertions?

- There are also a few situations where you should *not* use them:
- Do *not* use assertions for argument checking in public methods.
  - Argument checking is typically part of the published specifications (or *contract*) of a method, and these specifications must be obeyed whether assertions are enabled or disabled.
  - Erroneous arguments should result in an appropriate runtime exception (such as IllegalArgumentException, or NullPointerException). An assertion failure will not throw an appropriate exception.
- Do *not* use assertions to do any work that your application requires for correct operation.
  - Since assertions may be disabled, programs must not assume that the boolean expression contained in an assertion will be evaluated. Violating this rule has dire consequences.

Question time

Please try to limit the questions to the topics discussed during the session.

Participants are encouraged to discuss other issues during the breaks.

Thank you.