**final, static and others**

**final:**

The final keyword helps in making a variable unmodifiable after the initialization. If a variable is defined with final keyword, then it can not be modified after the initialization. This kind of check prevents the developers from accidentally modifying a variable, a method or a class.

* **Local variable:** If a variable is declared with the keyword final, then the value can not be changed after initialization. Here the variable pi is declared and initialized to 3.14, if we want to change it 3.45, then it causes a compilation error.

final double pi = 3.14;  
pi = 3.45; // Causes a compilation error

* **Member variable:** If a member variable of a class is marked as final, then it has to be initialized while it is declared or initialized in the constructor. Otherwise it causes a compilation error. Here the initializations in class A and B are correct, where as for class C, it is a problem since the variable j is neither initialized when declared nor initialized in the constructor.

class A  
{  
    final int i = 10;  
}  
  
class B  
{  
    final int k;  
      
    B(int k)  
    {  
        this.k = k;  
    }  
}  
  
class C  
{  
    final int j; // Causes a compilation error  
  
    C()  
    {  
    }  
}

* **Class or static variable:** If a static variable of a class is marked as final, then it has to be initialized while it is declared or initialized in the static block. Otherwise it causes a compilation error. Here the initializations in class M and N are correct, where as for class O, it is a problem since the variable j is neither initialized when declared nor initialized in the static block.

class M  
{  
    static final int i = 10;  
}  
  
class N  
{  
    static final int k;  
      
    static  
    {  
        k = 4 \* 20;  
    }  
}  
  
class O  
{  
    static final int j; // Causes a compilation error  
}

* **Method**: If a method of a class is marked as final, then it can not be overridden in any of its sub-classes. It causes a compilation error, if we try to override a final method.

class X  
{  
    final void print()  
    {  
        System.out.println("This method can not be overridden.");  
    }  
}  
  
class Y extends X  
{  
    void print() // Causes a compilation error, can not override a final method  
    {  
    }      
}

* **Class**: If a class itself is marked as final, then it can not be inherited (or extended). It causes a compilation error, if we try to extend a final class. Here class P is marked as final, hence it can not be inherited.

final class P  
{  
}  
  
class Q extends P  
{  
    // Causes a compilation error, can not inherit a final class  
}

**static:**

static keyword helps in declaring class variables. static variables can be used to define global variables.

As we know, with member variables of a class, if multiple objects are created, then every object will have its variable and changing variable of one object does not impact the variable of the other object.

class A  
{  
    int i;  
}  
...  
A a1 = new A();  
A a2 = new A();  
a1.i = 30;  
a2.i = 50;

It can be used to track number of objects created or some other usecase

class CountStudents  
{  
    public static void main(String s[])  
    {  
        Student st1 = new Student("Manohar", 34, 'A');  
        Student st2 = new Student("Uday", 78, 'B');  
        System.out.println("Number of students after st1, st2 : " + Student.count);  
        Student st3 = new Student("Kartik", 65, 'A');  
        System.out.println("Number of students after st3: " + Student.count);  
        Student st4 = createStudent();  
        System.out.println("Number of students after st4: " + Student.count);  
        System.out.println("Print count using objects : " + st1.count + " " + st2.count + " " + st3.count + " " + st4.count);  
    }  
      
    public static Student createStudent()  
    {  
        return new Student("New Student", 59, 'C');  
    }  
}  
  
class Student  
{  
    // Static variable  
    static int count = 0; // LINE A  
  
    // Member variables  
    String name;  
    int marks;  
    char section;  
  
    Student(String name, int marks, char section)  
    {  
        this.name = name;  
        this.marks = marks;  
        this.section = section;  
        // Increment the static variable  
        count++; // LINE B  
    }  
}

**Marking static variables as final**

The static variables of any class can be marked as final, so that they become constant or unmodifiable. Declaring the constants as public static final is a very common practice.

public static final double PI = 3.14;  
public static final int NUMBER\_OF\_BALLS\_IN\_A\_OVER = 6;

**Other uses - static methods**

**Static blocks:**

We can use static blocks inside a class to initialize static variables or do any one time activities, we want to perform before the class is used.

class Temperature  
{  
    public static final double FEET\_TO\_METER\_CONVERSION = 0.3048;  
    public static final double METER\_TO\_FEET\_CONVERSION;  
      
    static  
    {  
        METER\_TO\_FEET\_CONVERSION = 1 / FEET\_TO\_METER\_CONVERSION;  
        // Do other one time activities, like loading data from file, creating database connection pool etc.  
    }  
  
    static  
    {  
        // Do some more activities  
    }  
}

As shown above, we can have as many static blocks as needed and they will be called in the same order as they appear in the file. Also note that they will be called only once, no matter how many objects are created or how many times the class is accessed.

**Static methods:**

In any class, we can also define static methods. Similar to static variables, we can access static methods, using the class name and do not need any object for calling them.

When we create static methods as shown below, we need not create unnecessary new objects, every time we want to make a conversion.

class ConversionUtils  
{  
    public static double convertToFeet(double meters)  
    {  
        ...  
    }  
  
    public static double convertToMeters(double feet)  
    {  
        ...  
    }  
  
    public static double convertToFahrenheit(double celsius)  
    {  
        ...  
    }  
  
    public static double convertToCelsius(double fahrenheit)  
    {  
        ...  
    }  
}  
....  
double meters = ConversionUtils.convertToMeters(300.25);  
double fahrenheit = ConversionUtils.convertToFahrenheit(37.67);

**static and non-static methods**

we can have static methods and variables in a class. We can also have non-static methods and variables. These are nothing but the normal member variables and methods.

class A  
{  
    static int i\_static = 0;  
    int j\_non\_static;  
  
    public static void printStatic()  
    {  
        System.out.println("i\_static : " + i\_static);  
  
        // The below two statements causes compilation errors  
        // System.out.println("j\_non\_static : " + j\_non\_static);  
        // printNonStatic();  
    }  
  
    public void printNonStatic()  
    {  
        // All the three statements work fine.  
  
        System.out.println("i\_static : " + i\_static);  
        System.out.println("j\_non\_static : " + j\_non\_static);  
        printStatic();  
    }  
}

**The access rules are :**

* We can access a static variable from a static method. e.g., i\_static is accessed from printStatic() method.
* We CAN NOT access a non-static variable or non-static method from a static method. e.g., j\_non\_static and printNonStatic() can not be accessed from printStatic() method.
* We can access a non-static variable from a non-static method. e.g., j\_non\_static can be accessed from printNonStatic() method.
* We can also access a static variable or static method from a non-static method. e.g., i\_static and printStatic() can be accessed from printNonStatic() method.

**Understanding A simple Java Program**

class PrintHelloWorld  
{  
    public static void main(String args[])  
    {  
        System.out.println("Hello World");  
    }  
}

* **Why class:** In Java, every line of code/statement should be included as part of a class, that is the reason, we have defined a class with the name of the program PrintHelloWorld.
* **Why main:** When we run the program using java PrintHelloWorld it will look for the method main in the class PrintHelloWorld. The method main is the starting point of execution of the program.
* **Why static:** If we want to call a method, we should have an object, since we can not create an object without going into main method, it will become a problem. If we make the main method as static then we can call it with out creating any object.
* **Why public:** Since the main method should be accessible from outside of the program and probably from different packages, you need to make it public. If you make it private or protected or default, then the method can not be accessed from a different package.
* **Why void:** Even if return any parameter from the main method, the program does not know what to do with it. If you want to return a process exit code, we should use the method System.exit and the not return the value from main method.
* **Why String[]:** One way to pass parameters to java program, is to use the command line arguments. When we run the program with command line arguments, the arguments are converted into a String array and passed as a parameter to the main method.

**Exceptions:**

An *exception* is an abnormal condition that arises in a code sequence at run time. An exception is a run-time error.

When we compile using Java Compiler (javac) we get the compile-time errors, but when we run the program using (java) we get the run-time errors.

**Examples:**

int d = 0;  
int a = 42 / d;

int marks[] = new int[5];  
marks[6] = 77;

So, we need to handle these exceptions.

The best way to handle exception is by preventing it. Since exception causes abnormal termination of programs and they consume lot of resources compared the to the normal code execution

**Avoiding the divide by zero exception**

class DistributeChocolatesForStudents  
{  
    public static void main(String arg[])  
    {  
        int number\_of\_students = 5;  
        int number\_of\_chocolates = 15;  
          
        if( number\_of\_students <= 0 )  
        {  
            System.out.println("Chocolates can not be distributed since there are no students or negative number of students.");  
        }  
        else  
        {  
            int number\_of\_chocolates\_per\_student = number\_of\_chocolates / number\_of\_students;  
            System.out.println("Every student gets " + number\_of\_chocolates\_per\_student + " chocolates.");  
        }  
      
    }  
}

An *exception* is an abnormal condition that arises in a code sequence at run time. An exception is a run-time error. Most of these run-time errors could be prevented by doing proper input validation

Shown below is the simplest form of exception handling. Here we have two keywords try and catch. Just after the try keyword is the *try-block* and the block after the catch keyword is *catch-block*.

***try  
{  
    // This is try-block  
    // Include the statements which might cause an exception here  
}  
catch(Exception ex)  
{  
    // This is catch-block  
    // Handle exception here  
}***

In the *try-block*, we need include the statements which might cause exception, like opening of files, writing data into files or connecting to a remote web server or opening a database connection etc., In the *catch-block*, we can catch the exception and handle it. Usually in the catch-block we log the exceptions to the log file, so that the admin person knows about it. In the catch block, we will also inform the user of the program about the error in an understandable form.

**Divide by zero + array out of bounds example**

**Multiple catch blocks**

**Finally block**

Please note that finally keyword in exception handling is different from the final keyword

finally keyword is used in exception handling to include the code for releasing, closing or cleaning up of the resources.

**User Defined Exceptions**

Java provides a way to define our own exceptions. It is good practice to defined the custom(user-defined) exceptions for every application. This helps in handling exceptions better and pass additional information along with exception. The hierarchy of custom exceptions can help taking appropriate actions at various module levels.

class CustomException extends Exception  
{  
    String moreInfo;  
      
    CustomException(String moreInfo)  
    {  
        this.moreInfo = moreInfo;  
    }  
}

If we are dealing with railway reservation application, it might have custom exceptions like RailwayException, CounterClosedException, SeatsNotAvailableException, ServiceCancelledException, PaymentProcessingFailedException, InsufficientFundsException, PaymentGatewayNotRespondingException. And they could have a hierarchy as shown below.

**Throw and throws keyword**

Java supports two keywords throw and throws. throw helps in throwing a new exception or re-throwing a caught exception. throws needs to the placed after the method signature to indicate the caller of the method what exceptions a method can throw.

class TestRailwayExceptions  
{  
    public static void main(String arg[])  
    {  
        try  
        {  
            String travel\_date = "21/12/2012";  
            TicketBooker ticketBooker = new TicketBooker();  
            ticketBooker.bookTicket(17023, travel\_date, 3, 1);  
            System.out.println("Tickets booked successfully.");  
        }  
        catch(RailwayException re)  
        {  
            System.out.println("Railway booking failed. Reason : " + re.getErrorMessage());  
        }  
      
    }  
}  
  
class RailwayException extends Exception  
{  
    int trainNo;  
    String errorMessage;  
      
    RailwayException(int trainNo, String errorMessage)  
    {  
        this.trainNo = trainNo;  
        this.errorMessage = errorMessage;  
    }  
  
    String getErrorMessage()  
    {  
        return errorMessage;  
    }  
}  
  
class ServiceCancelledException extends RailwayException  
{  
    String date;  
  
    ServiceCancelledException(int trainNo, String date)  
    {  
        super(trainNo, "The service of train " + trainNo + " is cancelled on " + date);  
        this.date = date;  
    }  
}  
  
class SeatsNotAvailableException extends RailwayException  
{  
    String date;  
  
    SeatsNotAvailableException(int trainNo, String date)  
    {  
        super(trainNo, "There are no seats available for the train " + trainNo + " on " + date);  
        this.date = date;  
    }  
}  
  
  
class TicketBooker  
{  
    public void bookTicket(int trainNo, String date, int adults, int children)  
        throws RailwayException // LINE A  
    {  
  
        if(isServiceCancelled(trainNo, date))  
        {  
            throw new ServiceCancelledException(trainNo, date); // LINE B  
        }  
  
        if(areSeatsAvailable(trainNo, date, adults, children))  
        {  
            confirmBooking(trainNo, date, adults, children);  
        }  
        else  
        {  
            throw new SeatsNotAvailableException(trainNo, date); // LINE C  
        }  
    }  
  
    private boolean isServiceCancelled(int trainNo, String date)  
    {  
        // Code for checking if the service is cancelled  
        return false; // LINE D  
    }  
  
    private boolean areSeatsAvailable(int trainNo, String date, int adults, int children)  
    {  
        // Code here for checking if the seats are available  
        return false; // LINE E  
    }  
  
    private void confirmBooking(int trainNo, String date, int adults, int children)  
    {  
        // code here to confirm the booking  
    }  
}

**Diff between Error,Exception**

Java throws two types of exceptions - one is Error and the other is Exception. Both the classes Error and Exception extend from a common super-class Throwable.

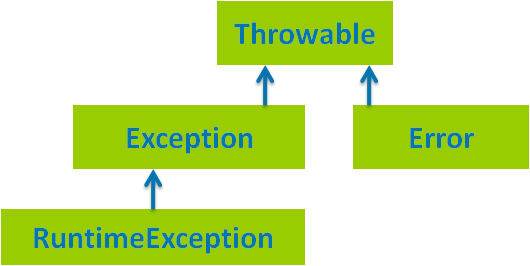
Any exception of type Exception can be handled by the user programs using the *try-catch* block

**IMPORTANT**

The RuntimeException class extends from Exception and any class which extends from RuntimeException is considered as *unchecked exception*. Any exception which extends from Exception class is considered as *checked exception*.

**Errors:**

The other type of exceptions are Error. These define exceptions that are not expected to be caught under normal circumstances by the user program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment. StackOverflowError is an example of Error. This is thrown when we recursively call the same method and it exceeds the number of method calls. OutOfMemoryError is also an Error and this is thrown by the run-time system, when it does not have sufficient memory to execute the program.

The hierarchy of Throwable, Exception, RuntimeException and Error is shown below. 

**Checked vs Unchecked Exceptions**

We discussed checked exceptions (extending the Exception class), for unchecked:

Java has one more special class called RuntimeException. We could also extend the custom exceptions from RuntimeException. Any class which extends directly from Exception is called checked exception, where as if it extends from RuntimeException it is called unchecked exception.

**NOTE:**

The main difference between checked exceptions and unchecked exceptions is, if a method throws a checked exception, then the calling method should have the code to catch it otherwise it will cause compile time error, where as if a method throws an unchecked exception, then it is up to the calling method whether to handle it or not.

**Examples:**

class CheckedException extends Exception  
{  
}  
  
class UncheckedException extends RuntimeException  
{  
}  
  
class A  
{  
    void method1() throws CheckedException  
    {  
    ...  
    }  
  
    void method2() throws UncheckedException  
    {  
    ...  
    }  
}

void callingMethod1()  
{  
    try  
    {  
        A a = new A();  
        a.method1();  
    }  
    catch(CheckedException ce)  
    {  
        // do some handling  
    }  
}

void callingMethod1() throws CheckedException  
{  
    A a = new A();  
    a.method1();  
}

// Causes compilation error.  
void callingMethod1()  
{  
    A a = new A();  
    a.method1();  
}

void callingMethod2()  
{  
    // Enclosing in try block is not mandatory  
    try  
    {  
        A a = new A();  
        a.method2();  
    }  
    catch(UncheckedException ce)  
    {  
        // do some handling  
    }  
}

// Rethrowing the exception is not mandatory  
void callingMethod2() throws UncheckedException  
{  
    A a = new A();  
    a.method2();  
}

void callingMethod2()  
{  
    A a = new A();  
    a.method2();  
}

Unless absolutely necessary it is suggested to use the unchecked exceptions, since the handling them or re-throwing them is not forced upon every method. This way we could have try-catch block at a very high level and save all other methods in between from handling or re-throwing them.

**Java Built in Exceptions – Checked and Unchecked**

**The list of unchecked exceptions or the exception classes which extend from RuntimeException.**

|  |  |
| --- | --- |
| **Exception** | **Description** |
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. More details at [Arraylist Access Using Index](http://java.meritcampus.com/core-java-topics/java-arraylist-access-by-index) |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. More details at [Assigning Super Class Reference To A Sub Class Reference In Java](http://java.meritcampus.com/core-java-topics/assigning-super-class-reference-to-a-sub-class-reference) |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. More details at [Member Variable In Java](http://java.meritcampus.com/core-java-topics/member-variable-in-java) |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| UnsupportedOperationException | An unsupported operation was encountered. |

**The list of checked exceptions.**

|  |  |
| --- | --- |
| **Checked Exceptions** | **Description** |
| ClassNotFoundException | Class not found. |
| CloneNotSupportedException | Attempt to clone an object that does not implement the Cloneable interface. |
| IllegalAccessException | Access to a class is denied. |
| InstantiationException | Attempt to create an object of an abstract class or interface. |
| InterruptedException | One thread has been interrupted by another thread. |
| NoSuchFieldException | A requested field does not exist. |
| NoSuchMethodException | A requested method does not exist. |

**Summary – Exception Handling**

* Every try should have a catch or finally block. We can not have a catch or finally block with out the try block.

// Valid - Try With Catch  
try  
{  
...  
}  
catch(Exception e)  
{  
}

// Valid - Try With Catch And Finally  
try  
{  
...  
}  
catch(Exception e)  
{  
...  
}  
finally  
{  
...  
}

// Valid - Try With Only Finally  
try  
{  
...  
}  
finally  
{  
...  
}

// Invalid - No Try Block  
catch(Exception e)  
{  
...  
}  
finally  
{  
...  
}

* A try can have multiple catch blocks but it can have only one finally block.

// Valid - try with multiple catch blocks and only one finally  
try  
{  
}  
catch(SubSubClassException e)  
{  
...  
}  
catch(SubClassException e)  
{  
...  
}  
catch(Exception e)  
{  
...  
}  
finally  
{  
...  
}

// INVALID - try with multiple finally blocks  
try  
{  
}  
catch(Exception e)  
{  
...  
}  
finally  
{  
...  
}  
finally  
{  
...  
}

* When we have multiple catch blocks for a try block, the sub-classes must be caught first and then the super-classes.

class AException extends Exception  
{  
...  
}  
class B1Exception extends AException  
{  
...  
}  
class B2Exception extends AException  
{  
...  
}

// Valid - Since the sub-sub-classes B1Exception, B2Exception are before the sub-class AException which is before the super-class Exception in the catch blocks  
try  
{  
}  
catch(B1Exception b1)  
{  
}  
catch(B2Exception b2)  
{  
}  
catch(AException a)  
{  
}  
catch(Exception e)  
{  
}

// INVALID - Since the sub-classes AException is before sub-sub-class B2Exception.  
try  
{  
}  
catch(AException a)  
{  
}  
catch(B2Exception b2)  
{  
}  
catch(Exception e)  
{  
}

// Valid - Since the sub-class AException is before super-class Exception. Although it does not include the sub-sub-classes.  
try  
{  
}  
catch(AException a)  
{  
}  
catch(Exception e)  
{  
}

// Valid - Since the sub-sub-class B1Exception is before super-class Exception. Although it does not include the sub-class AException.  
try  
{  
}  
catch(B1Exception b1)  
{  
}  
catch(Exception e)  
{  
}

* try blocks could be nested. We can have try blocks inside a catch block or inside a finally block.

try  
{  
    try  
    {  
        try{...}catch(Exception e){...}  
    }  
    catch(Exception e)  
    {  
        try{...}finally{...}  
    }  
}  
catch(Exception e)  
{  
    try{...}catch(Exception e){...}finally{ }  
}  
finally  
{  
    try{...}catch(Exception e){...}  
}

* throw keyword can be used to throw a new exception or re-throw a caught exception

try  
{  
    // Raise a new Exception  
    if(...) throw new AException();  
  
}  
catch(B1Exception b1)  
{  
    // Rethrow the caught exception after doing any required handling  
    throw b1;  
}

* In a method body, if we are throwing an exception, then that exception type (class name) should be included in the throws clause of the method. If there are more than one type of exception being thrown, then all the exceptions should be included or a common super-class of these exceptions can be included.

// Valid - since what ever exceptions being thrown are included in the throws clause  
void method() throws B1Exception, B2Exception  
{  
    if(...) throw new B1Exception;  
      
    if(...) throw new B2Exception;  
}

// Valid - since AException which is the super-class of B1Exception and B2Exception is included in the throws clause  
void method() throws AException  
{  
    if(...) throw new B1Exception;  
      
    if(...) throw new B2Exception;  
}

// INVALID - since the exceptions thrown are not included in the throws clause  
void method()  
{  
    if(...) throw new B1Exception;  
    if(...) throw new B2Exception;  
}

* If a method throws a *unchecked exception* or RuntimeException (or any class which extends from RuntimeException), then it is not necessary to include the calling of that method in a try block. But if it is a *checked exception* we need to include the calling of the method in try block and implement the exception handling in the catch block.

class YException extends RuntimeException  
{  
...  
}  
class Z1Exception extends YException  
{  
...  
}  
class Z2Exception extends YException  
{  
...  
}

// Valid - since Z1Exception and Z2Exception are run-time exceptions, they need not be included in the throws clause of the method  
void method()  
{  
    if(...) throw new Z1Exception  
    if(...) throw new Z2Exception  
}

// Valid - since Z1Exception and Z2Exception are run-time exceptions, there is no problem even if we include them in the throws clause of the method  
void method() throws Z1Exception, Z2Excetpion  
{  
    if(...) throw new Z1Exception  
    if(...) throw new Z2Exception  
}