

---

## Chapter 9

# Exception Handling

---

### What is in This Chapter ?

It is never possible to predict accurately what the user of your software will do. While your program is running, situations often arise in which some unexpected error occurs, perhaps due to unexpected or corrupt data. We have to deal with these problems gracefully in our code so that our code is robust, produces valid/correct results and does not crash. In this set of notes, we will discuss **Exceptions**, which are JAVA's way of handling problems that occur in your program. You will find out how to handle standard problems that occur in your code by using the **Exception** classes and how to define your own types of **Exceptions**.



## 9.1 Simple Debugging

We use the term **bug** in computer science to denote a problem with our program. Unfortunately, much of our programming time may be spent on finding errors/bugs in the code that we write. This can be VERY time consuming and frustrating. Sometimes we may fix one bug only to find that another one appears. There are basically 3 types of errors (i.e., bugs):

1. **Compile Errors** occur when your code will not compile. They are the easiest to since the compiler catches them and informs us of the problem. Because JAVA is strongly typed, many “misunderstandings” between method parameters and variables are eliminated. Once fixed, compiler errors do not come back. Often though ... one error (such as a missing semicolon) can lead to a whole slew of compile errors.
2. **Runtime Errors** cannot be determined at compile time. They “pop up” when you run your code and usually represent a serious problem (e.g., divide by zero, stack overflow, out of memory). These errors may sometimes require a re-design of your code (e.g., to reduce memory usage). But often, the problem is less serious such as trying to send messages to a **null** object (i.e., **NullPointerException**) or accessing past available array boundaries (i.e., **ArrayOutOfBoundsException**)
3. **Logic Errors** pertain to the logistics of your program such as computing wrong values or forgetting to handle certain “special situations” in your code. JAVA **cannot detect** nor **explain** these errors. Sometimes the logic error could lead to a runtime error which JAVA can then catch, but it certainly cannot explain them. Logic errors are often VERY difficult to find since the program could “appear” to be working. Rigorous testing is required to find them. Logic errors typically require you to do some *debugging*.



As programmers, we spend much of our time maintaining code and doing what is known as ...

**Debugging** is the processes of “figuring out errors” in your program and “fixing” them.



Actually, finding the error is usually the hard part. Fixing it is often (but certainly not always) easy. One of the most common debugging techniques is that of using “print” statements in your code. When there are many logic errors, this is usually the simplest way to debug.



If your program is producing wrong answers, you can use print statements to display intermediate calculations as follows ...

```
public double computeMortgagePayment() {
    double monthlyRate = this.getInterestRate() / 12;
    System.out.println("monthly rate = " + monthlyRate);    // debug

    double amortizeRate = (1-Math.pow(1+monthlyRate, this.numMonths*-1));
    System.out.println("amortize rate = " + amortizeRate); // debug

    return this.getHousePrice() * monthlyRate / amortizeRate;
}
```

From the intermediate results, you should be able to narrow down where you went wrong. Print statements can also be used to determine whether or not a certain point in your code is being reached (or if a certain method is being called):

```
public double computeMortgagePayment() {
    System.out.println("*** Got Here 1");
    double monthlyRate = this.getInterestRate() / 12;
    double amortizeRate = (1-Math.pow(1+monthlyRate, this.numMonths*-1));

    System.out.println("*** Got Here 2");

    return this.getHousePrice() * monthlyRate / amortizeRate;
}
```

By doing this, we can get an idea as to where our program has stopped working and also find out if JAVA is calling the methods that we think it is calling. Print statements can also be used to show the order that certain pieces of code are evaluated in:

```
public void deposit(float anAmount) {
    System.out.println("depositing $" + anAmount);
    this.balance = this.balance + anAmount;
}

public boolean withdraw(float anAmount) {
    System.out.println("withdrawing $" + anAmount);
    if (anAmount <= this.balance) {
        this.balance = this.balance - anAmount;
        return true;
    }
    return false;
}
```

In order to simplify the print statements, we can often print out whole objects ...

```
public static void Test1(){
    BankAccount account = new BankAccount( "Jim" );
    account.deposit(120.53f);
    account.withdraw(20);
    account.deposit(400);
    account.withdraw(829.31f);
    System.out.println(account);
}
```

As long as we have implemented an informative **toString()** method for our objects, we should get descriptive output.

Although this debugging technique is effective, your code may become littered with **System.out.println** statements which need to eventually be removed before you ship out your code. However, the "print statement" remains one of the most popular and simplest methods for debugging and this technique will usually help us narrow down the error that occurred.

In JAVA, it seems that the most common errors occur because we *forgot to initialize* something or if *unexpected* data was given to us. In some cases we can write additional code to "expect and handle" bad input data. This is called **error-checking** and it is the basis for **Exceptions** in JAVA. We will not discuss debugging any further in this course, but will instead focus on how to deal gracefully with unexpected errors that may arise in our programs.

## 9.2 Exceptions

There are many chances for errors to occur in a program when the programmer has no control over information that is entered into the program from the keyboard, files, or from other methods/classes/packages etc... Even worse ... when such errors occur, it is not always clear how to handle the error.



**Exceptions** are errors that occur in your program.

They are JAVA's way of telling you that something has gone wrong in your program. When an exception occurs, JAVA forces us to do one of the following:

- Handle the exception (we must know **when** to do this and **what** to do)
- Declare that we want **someone else** to handle it.

**Exception Handling** is the strategy of handling errors which are generated during program execution

We handle exceptions in order to allow our program to "quit gracefully" as opposed to having JAVA spew out a bunch of exception messages.

When should we handle exceptions ? When we do not know how to deal with the error ... or when it does not make sense to handle the error.

For example, in large software systems, an error may occur outside of the code that we wrote (i.e., in someone else's code). We may not even have access to this code in order to fix the error. Perhaps the error occurred in some module that was developed by another team of programmers. Sometimes, it is an advantage to anticipate some possible errors and then we can allow our program to handle the error gracefully. However, it is sometimes the case where we do not know what to do at all when the error occurs. If our code can easily predict a particular kind of error, then there is no need to use **Exceptions**, since we can deal with the code on our own.

Furthermore, in software components such as methods, libraries, and classes that are likely to be widely used, it is unclear as to what should be done when the error occurs. Our decision as to how we handle the error may or may not be the best choice for the software as a whole.

To help you understand, consider this "real world" example in which an unexpected situation occurs. Suppose that you ask your friend to go to McDonald's to get you a Big Mac and Fries. You expect him to come back with food for you.

However, what could go wrong ?

1. he crashes his car and never arrives at McDonald's
2. he gets there, but the place is burnt down
3. he places his order but finds out there are no Big Macs left anymore
4. he places the order but does not have enough money
5. he gets the food and drops/spills it on the ground on the way back



As you can see, much can go wrong ... but what would your friend do in each of these situations ?

1. he informs you that he cannot handle your request
2. he either returns informing you of the problem, or drives to a different McDonald's or nearby restaurant
3. he improvises and gets you two single hamburgers in the place of the Big Mac
4. he gets you an incomplete order
5. he tries to save money and simply wipes it off ;) ... or perhaps purchases replacements.

As you may well understand by now, we need to think along these lines. We need to always ask ourselves:

- What can go wrong ?
- Should I handle it ?
- How do I handle it ?

As it turns out, JAVA has a nice mechanism for handling errors in a consistent manner. We don't necessarily ever need to use this mechanism in our code, but there are advantages:

- **Improves clarity** of programs for large pieces of software
- Can be **more efficient** than "home-made" error checking code
- They apply to **multi-threaded** (more than one program) applications
- Programmers **save time** by using predefined Exceptions

In JAVA, exceptions are **thrown** (i.e., generated) by either:

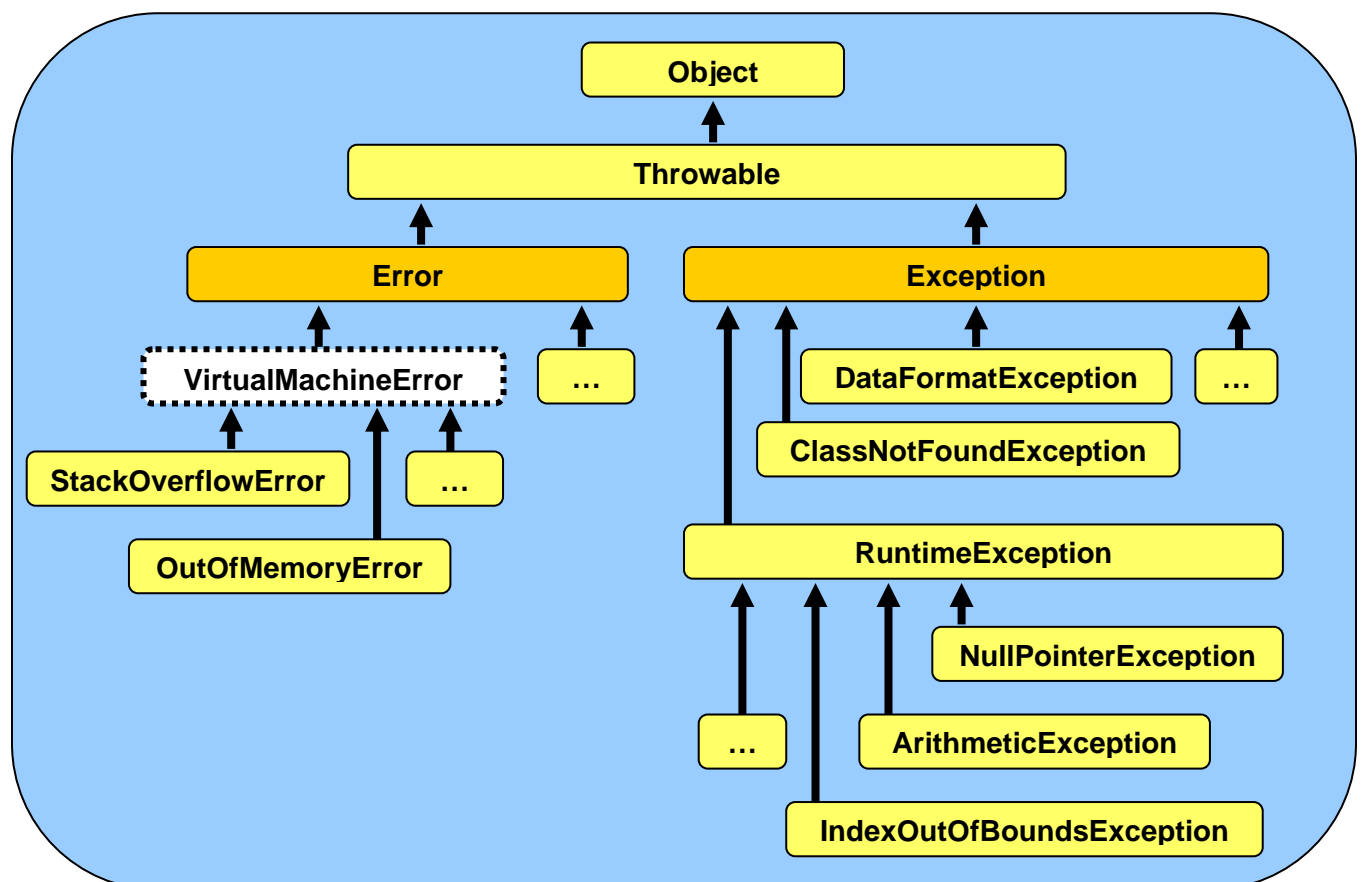
- the JVM automatically
- your code at any time

Exceptions are always **caught** (i.e., handled) by one of these:

- your own code (i.e., graceful decision)
- someone else's code (i.e., delegate the responsibility)
- the JVM (i.e., program halts)



JAVA has many predefined exceptions, and we can also create our own. In JAVA, **Exceptions** are objects, so each one is defined in its own class. The **Exception** classes are arranged in a hierarchy, and their position in the hierarchy can affect the way that they are handled. There are also **Error** objects in JAVA ... which represent more serious errors that may occur in your program which would require the program to stop altogether since they are considered unrecoverable:





In regards to the **Error** classes, generally your application should not try to catch them. There are many subclasses of **Error**, here are just a few:

- `VirtualMachineError`
  - `StackOverflowError` (e.g., recursion too deep)
  - `OutOfMemoryError` (e.g., can't create any more objects)
- `LinkageError`
  - `NoClassDefFoundError` (e.g., no class with given name)
  - `ClassFormatError` (e.g., class is incompatible)



The **Exception** class and its subclasses indicate a "less serious" problem. The exceptions are either "checked" or "unchecked" by the compiler. "Checked" exceptions are pre-defined types of errors that the JAVA compiler looks for in your code and forces you to deal with them before it will compile your code. Generally, your applications will need to deal with these types of **Exceptions**.

Here are just a few of the "checked" exceptions that we might need to catch in our code:

- `ClassNotFoundException` (e.g., tried to load an undefined class)
- `CloneNotSupportedException` (e.g., cannot make copy of object)
- `DataFormatException` (e.g., bad data conversion)
- `IllegalAccessException` (e.g., access modifiers prevent access)
- `InstantiationException` (e.g., problem creating an object)
- `IOException`
  - `EOFException` (e.g., end of file exception)
  - `FileNotFoundException` (e.g., cannot find a specified file)

Here are a few of the "unchecked" exceptions. Although you can check for (i.e., detect and handle) these types of errors in your code, normally you will not do so. Instead, you will try to write your code so that such exceptions cannot happen. The JAVA compiler will not force you to handle these errors before compiling:

- `RuntimeException`
  - `ArithmeticException` (e.g., bad computation such as divide by 0)
  - `ArrayStoreException` (e.g., storing wrong type of object in array)
  - `ClassCastException` (e.g., cannot typecast one class to another)
  - `IndexOutOfBoundsException` (e.g., gone outside array bounds)
  - `NoSuchElementException` (e.g., cannot find any more elements)
  - `NullPointerException` (e.g., attempt to send message to null)
  - `NumberFormatException` (e.g., trouble converting to a number)

Recall that when **exception handling** you must either (a) handle the exception yourself, or (b) declare that someone else will handle it.



In the 2<sup>nd</sup> case, we are actually **delegating** the exception-handling responsibility to someone else. We do this when we do not wish to handle an error in our code. We actually delegate the responsibility to the "calling method" (i.e., the method that called our method must handle the error). We do this by adding a *throws clause* to our method declaration as follows ...

```
public void openFile(String fileName) throws java.io.FileNotFoundException {  
    // code for method  
}
```

The **throws** keyword appears at the end of a method signature and is followed by an **Exception** type. When compiling this method, JAVA will check all methods that call this **openFile()** method to make sure that they deal with the **FileNotFoundException** in some way (i.e., either by catching it, or declaring that they too will throw it, thereby delegating the responsibility further up the chain of method calls).

You can actually specify multiple exception types with the **throws** clause by listing the exceptions separated by commas:

```
void convertFile(String fileName) throws java.io.FileNotFoundException,  
                                         java.lang.ClassNotFoundException,  
                                         java.io.IOException {  
    // code for method  
}
```

So, to clarify things a little, the **throws** clause is part of a method's declaration that is used to tell the compiler which exceptions the method may throw back to its caller. The **throws** clause is **required** if the code in the method "may" generate, but not handle, a particular type of exception. You should think of the **throws** clause as a "**sign**" that the method holds up in order to tell the whole world publicly that the code in that method may generate the specified exception.



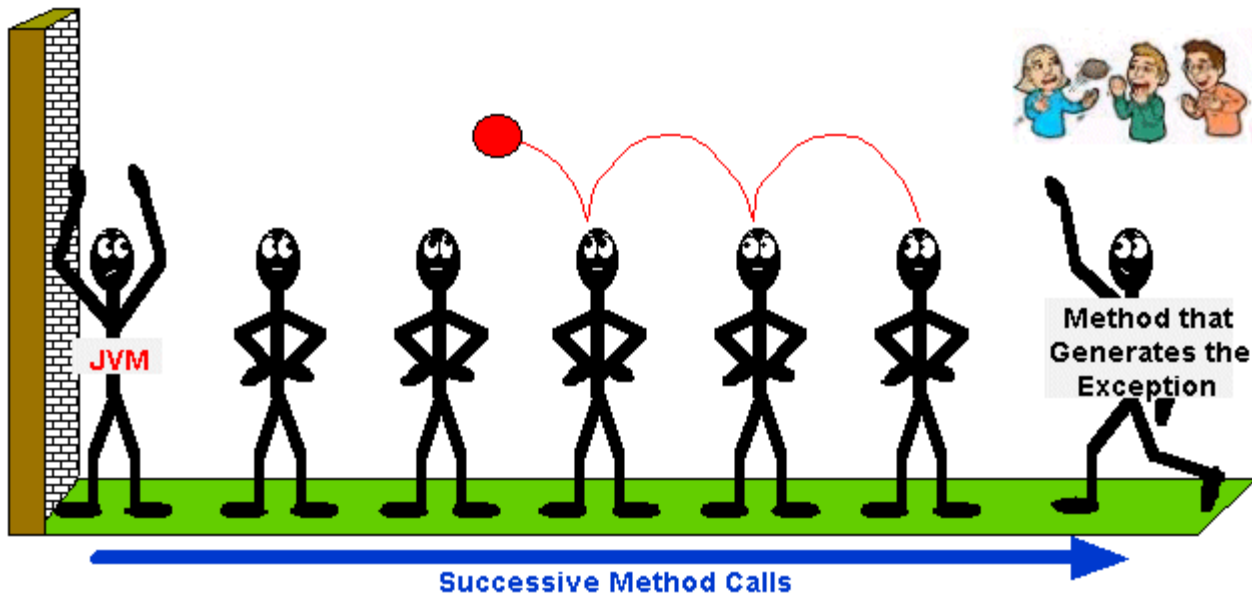
For example, if we consider the **openFile()** method mentioned earlier, it declares to everyone in its signature that it may generate a **FileNotFoundException** at any time. So, when we call the **openFile()** method from some other method, say **getCustomerInfo()**, then the **getCustomerInfo()** method "may also" declare that it throws the exception (if, for example, it did not want to handle it):

```
public void getCustomerInfo() throws java.io.FileNotFoundException {  
    // do something  
    this.openFile("customer.txt");  
    // do something  
}
```

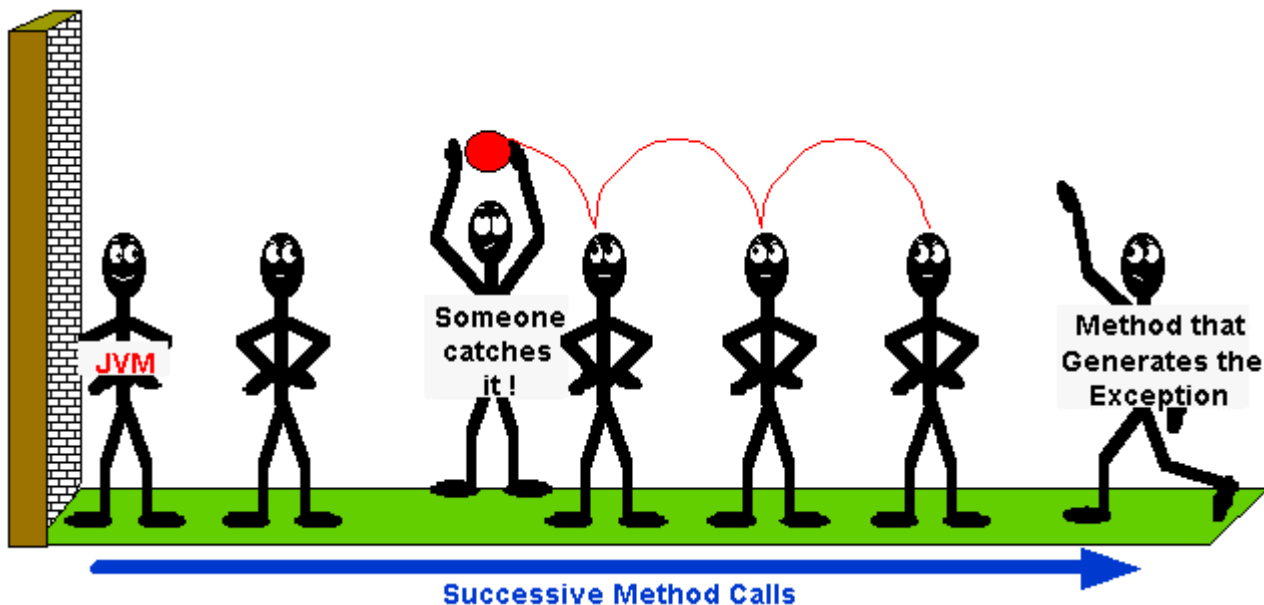
Here, if the exception is thrown while in the **openFile()** method, the **getCustomerInfo()** method will stop and it will then pass on the exception to "its" caller.



The responsibility may be repeatedly delegated in this manner. It is as if everyone ignores the error (like a hot potato). Nobody explicitly handles the error. The JVM will eventually catch it and halt the program:



At any time during this process however, any method may catch the exception and handle it. Once caught, propagation of the exception stops.



A method may catch an exception by specifying **try** and **catch** blocks. A "block" here refers to a sequence of JAVA statements (i.e., code defined between braces { }).

The "**try** block" represents the code for which you want to handle an Exception. We precede this block with the **try** keyword. Similarly, the "**catch** block" represents the code that handles a particular type of exception. We precede these blocks with the **catch** keyword.

A **catch** block always appears right after a **try** block as follows:

```
...
try {
    // some code that may cause an exception
}
catch (FileNotFoundException ex) {
    // some code that handles the exception
}
...
```

Notice that the **catch** block requires a parameter which indicates the type of error to be caught. This parameter can be accessed and used within the **catch** block (more on this later). The **getCustomerInfo()** method in our previous example can decide to handle the exception through use of **try/catch** blocks as follows:

```
public void getCustomerInfo() {
    try {
        this.openFile("customer.txt");
    }
    catch (java.io.FileNotFoundException ex) {
        System.out.println("Error: File not found"); // Handle the error here
    }
}
```

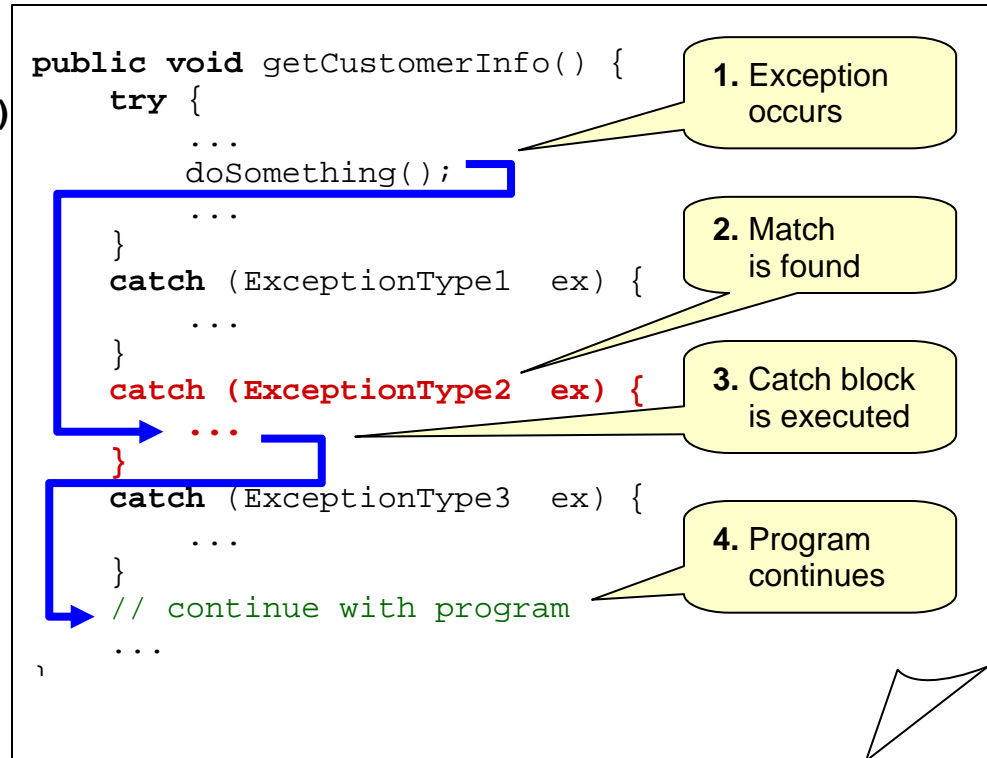
Notice that the method no longer needs to "throw" the exception any further (i.e., no **throws** clause), since it caught and handled it.

More than one **catch** block may be used to catch one-of-many possible exceptions. We simply list all **catch** blocks one after another:

```
public void getCustomerInfo() {
    try {
        // do something that may cause an Exception
    }
    catch (java.io.FileNotFoundException ex) {
        // Handle the error here
    }
    catch (NullPointerException ex) {
        // Handle the error here
    }
    catch (ArithmeticException ex) {
        // Handle the error here
    }
}
```

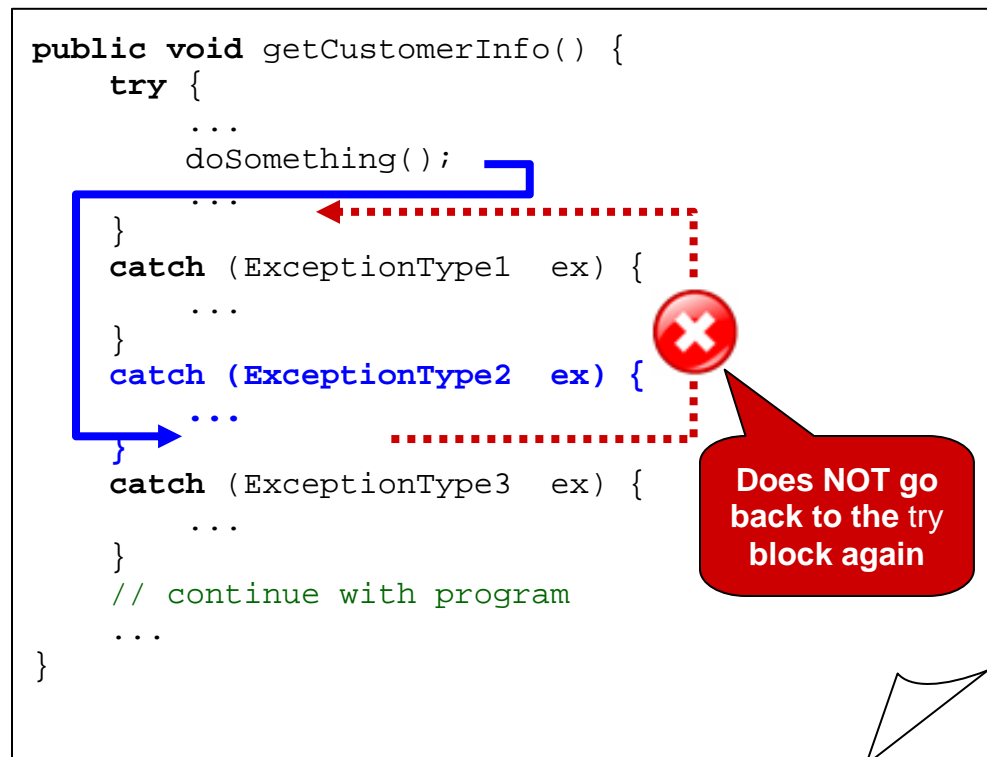
Consider what happens when an exception occurs within a **try** block:

Here, an exception of type **ExceptionType2** occurs as a result of the **doSomething()** method call. JAVA will immediately stop running the code in the **try** block and search through the **catch** blocks for one whose parameter type matches the exception that occurred (i.e., for one that takes **ExceptionType2** parameter or a superclass of **ExceptionType2**). When one is found, it executes the code within that **catch** block and then continues to the point in the program immediately following the **catch** blocks.



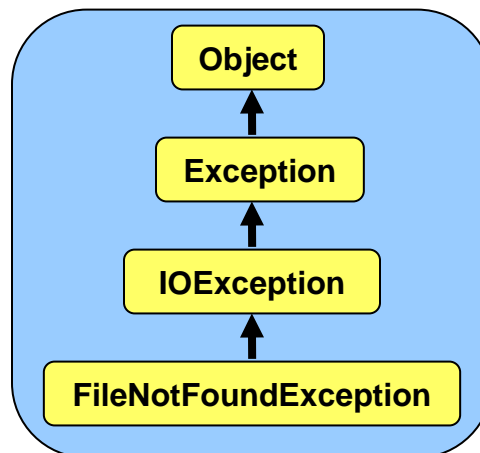
Note that JAVA does NOT go back to the **try** block once it completes the **catch** block. So any code remaining in the **try** block after the location where the exception had occurred is not evaluated as shown here →

If no match is found when JAVA looks for a matching **catch** block, then the entire **getCustomerInfo()** method halts and the method throws the same exception to the method that called this **getCustomerInfo()** method and that method will then have to deal with the exception in some way.



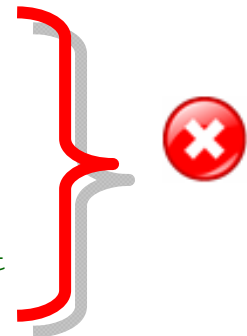
Since **Exceptions** are objects and are organized in a class hierarchy, then one **Exception** may be a more specific kind of another one. That is, **Exceptions** in general may have superclasses and subclasses. So, when JAVA goes looking through the **catch** blocks for a match, it will look for the first match that either matches the **Exception** class exactly or matches one of its superclasses. It is important to note that only one **catch** block (the one that "first" matches the exception) will ever be evaluated. That means we need to be careful, because the order of the **catch** blocks is important when we list them.

Consider the following portion of the JAVA class hierarchy:



The code below is problematic. Do you know why ?

```
public void getCustomerInfo() {
    try {
        // do something that may cause an exception
    }
    catch (Exception ex){
        // Catches all exceptions
    }
    catch (java.io.IOException ex){
        // Never reached since above catches all
    }
    catch (java.io.FileNotFoundException ex){
        // Never reached since above two are caught first
    }
}
```



Notice that we arranged the **catch** blocks so that the more general **Exception** is caught first. But this is bad because ALL exceptions are subclasses of **Exception**. That means, regardless of what type of exception occurs in the **try** block, the "first" **catch** block will ALWAYS match and therefore ALWAYS be evaluated. The remaining to **catch** blocks will never be evaluated. In fact, the JAVA compiler will detect this and tell you that the last two **catch** blocks are "unreachable". To fix the problem, we can simply reverse the order of the **catch** blocks.

Optionally, a special **finally** block may be used after the **catch** blocks:

```
try {  
    ...  
}  
catch (java.io.IOException ex){ ... }  
catch (Exception ex){ ... }  
finally {  
    // Code to release resources  
}
```

The **finally** block is used to release resources (e.g., closing files). It is always executed. That is, if no exception occurs, it is executed immediately after the **try** block, even if the **try** block has a **return** statement in it ! (i.e., it is executed just before returning). If an exception *does* occur, the **finally** block is executed immediately after the **catch** block is executed. If an exception occurs and no **catch** block matches, the **finally** block is evaluated before the method halts with the thrown exception.

Let us now look at what we can do inside our **catch** blocks. While inside the **catch** block, the following messages can be sent to the incoming **Exception** (i.e., to the parameter of a **catch** block):

- `getMessage()` - returns a **String** describing the exception. Typically, these strings are short descriptions of the error.
- `printStackTrace()` - displays the sequence of method calls that led up to the exception. This is what you see on the screen when the JVM catches an exception. This is very useful for debugging purposes.



So we can do many different things inside **catch** blocks. Here are some examples:

```
try {  
    ...  
}  
catch (ExceptionType1 ex) {  
    System.out.println("Hey!  Something bad just happened!");  
}  
catch (ExceptionType2 ex) {  
    System.out.println(ex.getMessage());  
}  
catch (ExceptionType3 ex) {  
    ex.printStackTrace();  
}
```

Consider the stack trace for this code:

```
import java.util.ArrayList;

public class MyClass {
    public static void doSomething(ArrayList<Integer> anArray){
        doAnotherThing(anArray);
    }
    public static void doAnotherThing(ArrayList<Integer> theArray){
        System.out.println(theArray.get(0));    // Error is generated
    }
    public static void main(String[] args){
        doSomething(null);
    }
}
```

When we run this code, we get the following stack trace printed to the console window:

```
java.lang.NullPointerException
  at MyClass.doAnotherThing(MyClass.java:7)
  at MyClass.doSomething(MyClass.java:4)
  at MyClass.main(MyClass.java:10)
```

Notice that the stack trace indicates:

1. the kind of **Exception** that was generated
2. the method that generated the exception and
3. the line number at which the exception occurred

## 9.3 Examples of Handling Exceptions

Let us now look at how we can handle (i.e., catch) a standard **Exception** in JAVA. Consider a program that reads in two integers and divides the first one by the second and then shows the answer. We will assume that we want the number of times that the second number divides evenly into the first (i.e., ignore the remainder). What problems can occur? Well, we may get invalid data or we may get a divide by zero error. Let us look at how we would have done this previously ...





```
import java.util.Scanner;

public class ExceptionTestProgram1 {
    public static void main(String[] args) {
        int        number1, number2, result;
        Scanner     keyboard;

        keyboard = new Scanner(System.in);
        System.out.println("Enter the first number:");
        number1 = keyboard.nextInt();

        System.out.println("Enter the second number:");
        number2 = keyboard.nextInt();

        System.out.print(number2 + " goes into " + number1);
        System.out.print(" this many times: ");

        result = number1 / number2;
        System.out.println(result);
    }
}
```

Here is the output if **143** and **24** are entered:

```
Enter the first number:
143
Enter the second number:
24
24 goes into 143 this many times: 5
```

What if we now enter **143** and **ABC** ?

```
Enter the first number:
143
Enter the second number:
ABC
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:840)
    at java.util.Scanner.next(Scanner.java:1461)
    at java.util.Scanner.nextInt(Scanner.java:2091)
    at java.util.Scanner.nextInt(Scanner.java:2050)
    at ExceptionTestProgram1.main(ExceptionTestProgram1.java:13)
```

This is not a pleasant way for your program to end. By default, when exceptions occur, they actually print out the stack trace (i.e., the sequence of method calls that led to the exception). That is what we are seeing here. It is ugly, but good for debugging purposes.

Notice what happened. The first line of the stack trace indicates that an **InputMismatchException** has occurred. The second line tells us that the error occurred at

line **840** of the **Scanner.java** code from a method called **throwFor()**. This was not code that we wrote ... it is pre-existing code from JAVA's **Scanner** class. The error, however, is not in line 840 of the **Scanner** class code. That is just where the error *surfaced*.

By looking further down the stack trace, we can gain insight as to why our code caused the **Exception** to occur. We just need to look down the stack trace until we find a method that we wrote. Notice that most of the successive method calls were in the **Scanner** class. However, right at the bottom we notice that the **main** method was called.

As it turns out, JAVA is telling us that the error occurred as a result of line **13** in our **ExceptionTestProgram1**. That is the code that tries to obtain the next integer from the **Scanner**. When it attempts to do this, we get an "Input Mismatch" because we entered ABC when we ran the program ... and ABC cannot be converted to an integer.

So now that we know WHY the error occurred, how can we gracefully handle the error ? We certainly do not want to see the stack trace message !!!

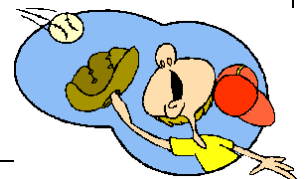
In order to handle the entering of bad data (e.g., ABC instead of an integer) we would need to do one of two things:

1. either modify the code in the **Scanner** class to detect and gracefully handle the error, or
2. catch the **InputMismatchException** within our code and gracefully handle the error.

Since it is not usually possible nor recommended to copy and start modifying the available JAVA class libraries, our best choice would be to catch and handle the error from our own code. We will have to "look for" (i.e., catch) the **InputMismatchException** by placing **try/catch** blocks in the code appropriately as follows:

```
try {
    System.out.println("Enter the first number:");
    number1 = keyboard.nextInt();

    System.out.println("Enter the second number:");
    number2 = keyboard.nextInt();
}
catch (java.util.InputMismatchException e) {
    System.out.println("Those were not proper integers! I quit!");
    System.exit(-1);
}
System.out.print(number2 + " goes into " + number1);
...
```



Notice in the **catch** block that we display an error message when the error occurs and then we do: `System.exit(-1);`. This is a quick way to halt the program completely.

The value of **-1** is somewhat arbitrary but when a program stops we need to supply some kind of integer value. Usually the value is a special code that indicates what happened. Often, programmers will use **-1** to indicate that an error occurred.

Once we incorporate the **try** block, JAVA indicates to us the following compile errors:

```
variable number2 might not have been initialized
variable number1 might not have been initialized
```

It is referring to this line:

```
System.out.print(number2 + " goes into " + number1);
```

Here we are using the **number1** and **number2** variables. However, because the **try** block may generate an error, JAVA is telling us that there is a chance that we will never assign values to these variables (i.e., they might not be initialized) and so we might obtain wrong data. JAVA does not like variables that have no values ... so it is forcing us to assign a value to these two variables. It is perhaps the most annoying type of compile error in JAVA, but nevertheless we must deal with it. The simplest way is to simply assign a value of 0 to each of these variables when we declare them. Here is the updated version:

```
import java.util.Scanner;

public class ExceptionTestProgram2 {
    public static void main(String[] args) {
        int        number1 = 0, number2 = 0, result;
        Scanner     keyboard;

        keyboard = new Scanner(System.in);
        try {
            System.out.println("Enter the first number:");
            number1 = keyboard.nextInt();

            System.out.println("Enter the second number:");
            number2 = keyboard.nextInt();
        }
        catch (java.util.InputMismatchException e) {
            System.out.println("Those were not proper integers! I quit!");
            System.exit(-1);
        }
        System.out.print(number2 + " goes into " + number1);
        System.out.print(" this many times: ");

        result = number1 / number2;
        System.out.println(result);
    }
}
```

If we test it again with **143** and **24** as before, it still works the same. However, now when tested with **143** and **ABC**, here is the output:

```
Enter the first number:
143
Enter the second number:
ABC
Those were not proper integers! I quit!
```

What if we enter **ABC** as the first number ?

```
Enter the first number:
ABC
Those were not proper integers! I quit!
```

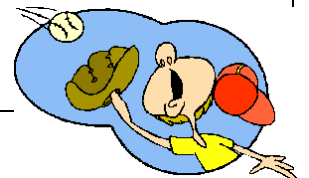
Woops! It appears that our error message is not grammatically correct anymore. Perhaps we should change it to **"Invalid integer entered!"** ... this should be clear enough.

Now lets test the code with values **12** and **0**:

```
Enter the first number:
12
Enter the second number:
0
0 goes into 12 this many times: Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at ExceptionTestProgram2.main(ExceptionTestProgram2.java:23)
```

JAVA has detected that we tried to divide a number by zero ... a big “no no” in the world of mathematics. We can handle the **ArithmeticException** by adding additional **try/catch** blocks around line **23** of our code:

```
try {
    result = number1 / number2;
}
catch (ArithmeticException e) {
    System.out.println("Second number is 0, cannot do division!");
    System.exit(-1);
}
System.out.println(result);
```



We can merge the two **try** blocks into one if we want to as follows...

```

import java.util.Scanner;

public class ExceptionTestProgram3 {
    public static void main(String[] args) {
        int        number1 = 0, number2 = 0, result = 0;
        Scanner    keyboard;

        keyboard = new Scanner(System.in);
        try {
            System.out.println("Enter the first number:");
            number1 = keyboard.nextInt();

            System.out.println("Enter the second number:");
            number2 = keyboard.nextInt();

            result = number1 / number2;
        }
        catch (java.util.InputMismatchException e) {
            System.out.println("Invalid integer entered!");
            System.exit(-1);
        }
        catch (ArithmeticException e) {
            System.out.println("Second number is 0, cannot do division!");
            System.exit(-1);
        }
        System.out.print(number2 + " goes into " + number1);
        System.out.println(" this many times: " + result);
    }
}

```

Now when we enter **12** and **0** as input, we get the appropriate message:

```
Second number is 0, cannot do division!
```

How can we adjust our code to repeatedly prompt for integers until valid ones were entered ?

We would need a **while** loop since we do not know how many times to keep asking.

Here is how we could do this to get a single number ...

```

int        number1 = 0;
boolean    gotANumber = false;

while (!gotANumber) {
    try {
        System.out.println("Enter the first number");
        number1 = new Scanner(System.in).nextInt();
        gotANumber = true;
    }
    catch (java.util.InputMismatchException e) {
        System.out.println("Invalid integer. Please re-enter");
    }
}

```

This code would repeatedly ask for a number until it was a valid integer. However, there is a slight problem with the **Scanner** class. When the error is generated in the **Scanner** class code due to the invalid integer being entered, the **Scanner** object is messed up and is no longer ready read integers using **nextInt()**. The easiest way to fix this is to re-assign a new **Scanner** object to the **keyboard** variable when the error occurs. Here is the completed code:

```
import java.util.Scanner;

public class ExceptionTestProgram4 {
    public static void main(String[] args) {
        int        number1 = 0, number2 = 0, result = 0;
        boolean     gotANumber = false;
        Scanner     keyboard;

        keyboard = new Scanner(System.in);
        while(!gotANumber) {
            try {
                System.out.println("Enter the first number");
                number1 = keyboard.nextInt();
                gotANumber = true;
            }
            catch (java.util.InputMismatchException e) {
                System.out.println("Invalid integer. Please re-enter");
                keyboard = new Scanner(System.in);
            }
        }
        gotANumber = false;
        while(!gotANumber) {
            try {
                System.out.println("Enter the second number");
                number2 = keyboard.nextInt();
                gotANumber = true;
            }
            catch (java.util.InputMismatchException e) {
                System.out.println("Invalid integer. Please re-enter");
                keyboard = new Scanner(System.in);
            }
        }
        try {
            result = number1 / number2;
            System.out.print(number2 + " goes into " + number1);
            System.out.println(" this many times: " + result);
        }
        catch (ArithmeticException e) {
            System.out.println("Second number is 0, cannot do division!");
        }
    }
}
```



Here are the test results:

```
Enter the first number
what
Invalid integer. Please re-enter
Enter the first number
help me
Invalid integer. Please re-enter
Enter the first number
ok, ok, here goes
Invalid integer. Please re-enter
Enter the first number
143
Enter the second number
did you say number 2 ?
Invalid integer. Please re-enter
Enter the second number
40
40 goes into 143 this many times: 3
```

## 9.4 Creating and Throwing Your Own Exceptions

You may throw an exception in your code at any time if you wish to inform everyone that an error occurred in **your** code. Thus, you do not need to *handle* the error in your code, you can simply *delegate* (i.e., transfer) the responsibility to whoever calls your method.

Exceptions are thrown with the **throw** statement. Basically, when we want to generate an exception, we create a new **Exception** object by calling one of its constructors, and then **throw** it as follows:

```
throw new java.io.FileNotFoundException();
throw new NullPointerException();
throw new Exception();
```

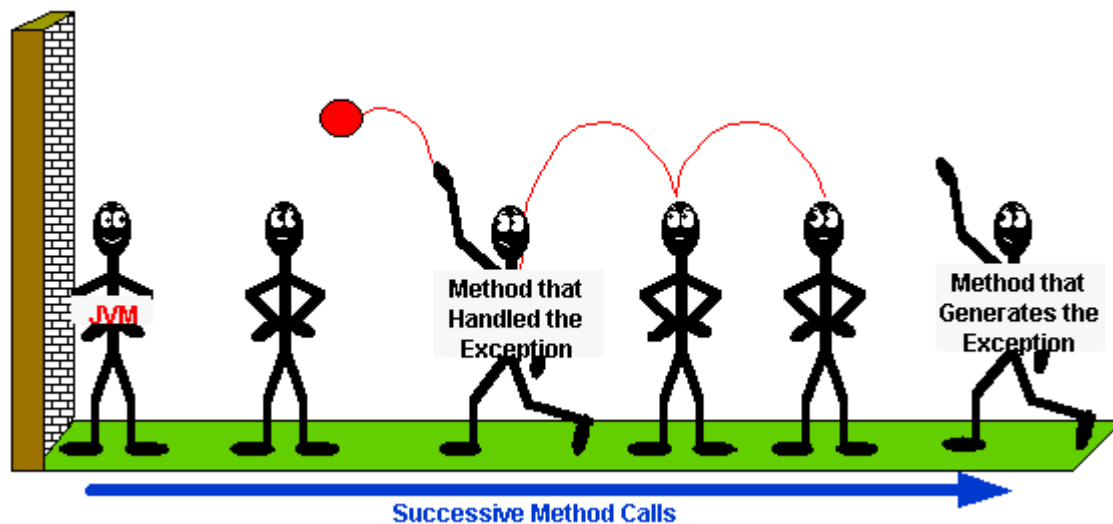
Methods that throw these exceptions, must declare that they do so in their method declarations, using the **throws** clause (as we have seen before):

```
public void yourMethod() throws anException {
    ...
}
```



You may even catch an exception, partially handle it and then throw it **again**:

```
public void yourMethod() throws Exception {  
    try {  
        ...  
    }  
    catch (Exception ex){  
        ...           // partially handle the exception here  
        throw ex;      // then throw it again  
    }  
}
```



Catching and then throwing an exception again is useful, for example, if we want to:

- just keep an internal "log" of errors that were generated
- attach additional information to the exception message
- delay the passing on of the exception to the calling method

It is also possible to create "your own" types of exceptions. This would allow you to catch specific types of problems in your code that JAVA would normally ignore. To make your own exceptions, you simply need to create a subclass of an existing exception. If you are unsure where to put it in the hierarchy, you should use **Exception** as the superclass.

Here are the steps to making your own **Exception**:

1. choose a *meaningful* class/exception name (e.g., **WrongPasswordException**)
2. specify the superclass under which this exception will reside (e.g., **Exception**)
3. optionally provide a constructor (for simplicity, this constructor may just call the **super** constructor, passing in a string indicating the reason for the error).

Here is an example of a newly defined exception called **MyExceptionName**. We define it just as we would any other class and then save it to a file called **MyExceptionName.java**. It must also be compiled before it can be used in your program ...

```
public class MyExceptionName extends Exception {  
    public MyExceptionName() {  
        super("Some string explaining the exception");  
    }  
}
```

Consider an example of how we could force the user to type in their name (i.e., not leave it blank). We could do the following:

```
import java.util.Scanner;  
  
public class MyExceptionTestProgram {  
    public static void main(String[] args) {  
        String    name = "";  
        boolean    gotValidName = false;  
        Scanner    keyboard = new Scanner(System.in);  
  
        while (!gotValidName) {  
            System.out.println("Enter your name");  
            name = keyboard.nextLine();  
            if (name.length() > 0)  
                gotValidName = true;  
            else  
                System.out.println("Error: Name must not be blank");  
        }  
        System.out.println("Hello " + name);  
    }  
}
```

Here would be the output of such a program:

```
Enter your name  
  
Error: Name must not be blank  
Enter your name  
Mark  
Hello Mark
```

Notice how the “error” is detected ... we simply check the data for an empty string and use **if/else** statements to determine whether or not the error has occurred and then display an appropriate message.

In some programs, however, we may not want to print a message to the screen. For example, we may want to bring up a dialog box. In fact, we may not know exactly what to do, as it depends on our user interface as well as the context within our application. In such cases (i.e., when we are not sure what to do), we could simply generate an exception and let the method that called our code decide what to do.

Lets generate a **MissingNameException** when the user does not enter a name. We can do this by starting with our own exception definition as follows:

```
public class MissingNameException extends Exception {  
    public MissingNameException() {  
        super("Name is blank");  
    }  
}
```

We need to save and compile that code in its own file. Now, how do we generate the exception? We simply call **throw new** `MissingNameException()` at the right spot in the code:

```
import java.util.Scanner;  
  
public class MyExceptionTestProgram2 {  
    public static void main(String[] args) throws MissingNameException {  
        String    name = "";  
        boolean    gotValidName = false;  
        Scanner    keyboard = new Scanner(System.in);  
  
        while (!gotValidName) {  
            System.out.println("Enter your name");  
            name = keyboard.nextLine();  
            if (name.length() <= 0)  
                throw new MissingNameException();  
            gotValidName = true;  
        }  
        System.out.println("Hello " + name);  
    }  
}
```

Notice that we must declare in our method that we now throw the exception. If we run the code as before, we can see this new exception being generated:

```
Enter your name
```

```
Exception in thread "main" MissingNameException: Name is blank  
    at MyExceptionTestProgram2.main(MyExceptionTestProgram2.java:12)
```

Congratulations to us ... we have successfully created and generated our own exception. How though can we handle the exception? So that we may use the same example, let us adjust the code a little by creating a method that will get the user input for us as follows ...

```
public String getName() throws MissingNameException {
    String name = new Scanner(System.in).nextLine();
    if (name.length() <= 0)
        throw new MissingNameException();
    return name;
}
```

The above method gets the name from the user and returns it ... unless the name is blank ... in which case it generates the **MissingNameException**.

Now we should catch the error from our **main** program as follows:

```
import java.util.Scanner;

public class MyExceptionTestProgram3 {

    // Method to get the name from the user
    public static String getName() throws MissingNameException {
        String name = new Scanner(System.in).nextLine();
        if (name.length() <= 0)
            throw new MissingNameException();
        return name;
    }

    // Main method to test out the MissingNameException
    public static void main(String[] args) {
        String name = "";
        boolean gotValidName = false;

        while (!gotValidName) {
            System.out.println("Enter your name");
            try {
                name = getName();
                gotValidName = true;
            }
            catch (MissingNameException ex) {
                System.out.println("Error: Name must not be blank");
            }
        }
        System.out.println("Hello " + name);
    }
}
```



The resulting output is the same as before (i.e., same as `MyExceptionTestProgram`).

As another example, let us take another look at the **BankAccount** object again ... more specifically ... consider this **withdraw()** method:

```
public boolean withdraw(float anAmount) {  
    if (anAmount <= this.balance) {  
        this.balance -= anAmount;  
        return true;  
    }  
    return false;  
}
```



When the user tries to withdraw more money than is actually in the account ... nothing happens. Since the method returns a **boolean**, we can always check for this error where we call the method:

```
public static void main(String[] args) {  
    BankAccount b = new BankAccount("Bob");  
    b.deposit(100);  
    b.deposit(500.00f);  
  
    if (!b.withdraw(25.00f))  
        System.out.println("Error withdrawing money from account");  
    if (!b.withdraw(189.45f))  
        System.out.println("Error withdrawing money from account");  
    b.deposit(100.00f);  
    if (!b.withdraw(1000000))  
        System.out.println("Error withdrawing money from account");  
}
```

This form of error checking works fine, but it clearly clutters up the code! Let us see how we can make use of an Exception. We will create a **WithdrawalException** object. Where would it go in the **Exception** hierarchy? Probably right under the **Exception** class again, since there are no existing bank-related exception classes in JAVA. Here is the exception:

```
public class WithdrawalException extends Exception {  
    public WithdrawalException() {  
        super("Error making withdrawal");  
    }  
}
```

Now how do we *throw* the exception from within the **withdraw()** method? Here is how we do it ...



```
public void withdraw(float anAmount) throws WithdrawalException {
    if (anAmount <= this.balance)
        this.balance -= anAmount;
    else
        throw new WithdrawalException();
}
```

Note that we must also instruct the compiler that this method may throw a **WithdrawalException** by writing this as part of the method declaration. The addition of this simple statement will force all methods that call the **withdraw()** method to deal with the exception.

Also notice that we no longer need the **boolean** return type for the **withdraw()** method since its purpose was solely for error checking. Now that we have the exception being generated, this becomes our new form of error checking.

Now how do we change the code that calls the **withdraw()** method ? We just need to enclose our withdrawal code in a **try** block:

```
public static void main(String[] args) {
    BankAccount b = new BankAccount("Bob");
    try {
        b.deposit(100);
        b.deposit(500.00f);
        b.withdraw(25.00f);
        b.withdraw(189.45f);
        b.deposit(100.00f);
        b.withdraw(1000000);
    } catch (WithdrawalException ex) {
        System.out.println("Error withdrawing money");
    }
}
```

Notice how much simpler and cleaner the calling code becomes. Be aware however, that if one error occurs early within the **try** block, none of the remaining code in the **try** block gets evaluated!!! So an error in the first withdrawal attempt would prevent the two other withdrawals and deposit being made on the account from happening. If we did not want this behavior, we would need to make a separate **try/catch** block for each of the 3 **withdraw()** method calls.

We can make our code even simpler by ignoring the error. To do this we would have to indicate in the **main()** method that the **WithdrawalException** may occur as follows ...

```
public static void main(String[] args) throws WithdrawalException {
    BankAccount b = new BankAccount("Bob");
    b.deposit(100);
    b.deposit(500.00f);
    b.withdraw(25.00f);
    b.withdraw(189.45f);
    b.deposit(100.00f);
    b.withdraw(1000000);
}
```

If we do this, however, then the program will stop and quit when the first **WithdrawalException** occurs.

We can actually add more information to our exceptions. For example, there may be many reasons why we cannot withdraw from a **BankAccount**. The bank account ...

- may not have enough money in it,
- may not allow withdrawals (e.g., some kinds of **SavingsAccounts**), or
- may not have sufficient funds to cover transaction fees associated with the account

We do not need to make different types of exceptions, but can instead supply more information when the **WithdrawException** is generated. The easiest way to do this is to modify the constructor in our **WithdrawalException** class that takes a **String** parameter to describe the error:

```
public class WithdrawalException extends Exception {
    public WithdrawalException(String description) {
        super(description);
    }
}
```

We can then use this new constructor instead by supplying different explanations as to why the error occurred.

For example, the **SuperSavings** account may have the following **withdraw()** method:

```
public void withdraw(float anAmount) throws WithdrawalException {
    throw new WithdrawalException("Withdrawals not allowed from this account");
}
```

whereas the **PowerSavings** account may have this method ...

```
public void withdraw(float anAmount) throws WithdrawalException {  
    if (anAmount > this.balance)  
        throw new WithdrawalException("Insufficient funds in account to  
                                        withdraw specified amount");  
    if (anAmount + WITHDRAW_FEE > this.balance) {  
        throw new WithdrawalException("Not enough money to cover  
                                        transaction fee");  
    }  
    this.balance -= anAmount + WITHDRAW_FEE;  
}
```

So, as can easily be seen, we can provide additional explanatory information for the user when an exception does occur. Furthermore, we can do this with a single exception class (i.e., we do not need to make a subclass of **WithdrawalException** for each specific situation).

We can extract this “additional explanation” from the Exception by sending the **getMessage()** message to the exception within our catch blocks:

```
public static void main(String[] args) {  
    PowerSavings p = new PowerSavings("Bob");  
    SuperSavings s = new SuperSavings("Betty");  
    try {  
        p.deposit(100);  
        s.deposit(500.00f);  
        p.withdraw(25.00f);  
        p.withdraw(189.45f);  
        s.deposit(100.00f);  
        s.withdraw(1000000);  
    } catch (WithdrawalException ex) {  
        System.out.println(ex.getMessage());  
    }  
}
```



In this example, the **catch** block catches any errors for both bank accounts.