# **Understanding Exceptions:**

An exception is Java’s way of saying, “I give up. I don’t know what to do right now. You deal with it.” When you write a method, you can either deal with the exception or make it the calling code’s problem. These are the two approaches Java uses when dealing with exceptions. A method can handle the exception case itself or make it the caller’s responsibility. The key point to remember is that exceptions alter the program flow.

## **Return Codes vs. Exceptions:**

Exceptions are used when “something goes wrong.” However, the word “wrong” is subjective. The following code returns –1 instead of throwing an exception if no match is found:

*public int indexOf(String[] names, String name) {*

*for (int i = 0; i < names.length; i++) {*

*if (names[i].equals(name)) { return i; }*

*}*

*return -1;*

*}*

This approach is common when writing a method that does a search. For example, imagine being asked to find the name Joe in the array. It is perfectly reasonable that Joe might not appear in the array. When this happens, a special value is returned. An exception should be reserved for exceptional conditions like names being null.

In general, try to avoid return codes. Return codes are commonly used in searches, so programmers are expecting them. In other methods, you will take your callers by surprise by returning a special value. An exception forces the program to deal with them or end with the exception if left unhandled, whereas a return code could be accidentally ignored and cause problems later in the program. An exception is like shouting, “Deal with me!”

As we’ve explained, an exception is an event that alters program fl ow. Java has a Throwable superclass for all objects that represent these events.

## Errors:

Error means something went so horribly wrong that your program should not attempt to recover from it. For example, the disk drive “disappeared.” These are abnormal conditions that you aren’t likely to encounter.

## Unchecked Exceptions:

A runtime exception is defined as the RuntimeException class and its subclasses. Runtime exceptions tend to be unexpected but not necessarily fatal. For example, accessing an invalid array index is unexpected. Runtime exceptions are also known as unchecked exceptions.

## Checked Exceptions:

A checked exception includes Exception and all subclasses that do not extend RuntimeException. Checked exceptions tend to be more anticipated—for example, trying to read a file that doesn’t exist.

Checked exceptions? What are we checking? Java has a rule called the handle or declare rule. For checked exceptions, Java requires the code to either handle them or declare them in the method signature.

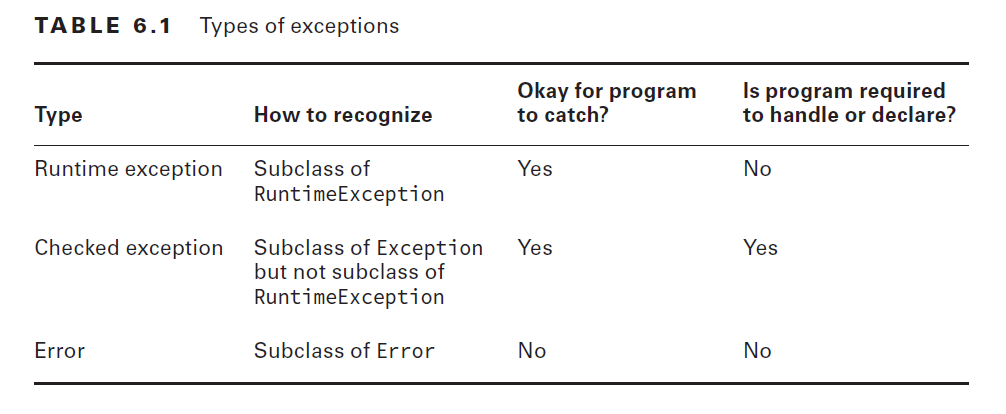
For example, this method declares that it might throw an exception:

*void fall() throws Exception {*

*throw new Exception();*

*}*

* throw tells Java that you want to throw an Exception.
* throws simply declares that the method might throw an Exception.



# **Using a try Statement:**

Now that you know what exceptions are, let’s explore how to handle them. Java uses a try statement to separate the logic that might throw an exception from the logic to handle that exception.

# 

# try statements are like methods in that the curly braces are required even if there is only one statement inside the code blocks.

# **Try block** can not be used alone it needs to be paired with **catch** or **finally** block or both.

# If an exception is thrown in try block, the code in catch block is executed first than code in finally block is executed.

# And if If an exception is not thrown in try block, the code in catch block skipped but the code in finally block is executed.

# The order in which block appears matter a lot, it should be only in this order try→catch→finally.

# There is one exception to “the finally block always runs after the catch block” rule: Java defines a method that you call as System.exit(0);. The integer parameter is the error code that gets returned. System.exit tells Java, “Stop. End the program right now. Do not pass go. Do not collect $200.” When System.exit is called in the try or catch block, finally does not run.

# A rule exists for the order of the catch blocks. Java looks at them in the order they appear. If it is impossible for one of the catch blocks to be executed, a compiler error about unreachable code occurs. This happens when a superclass is caught before a subclass.

# If an exception is thrown in a catch block and also in finally block. Java will forget about the exception thrown in catch block and exception from finally block will be thrown.

# **Recognizing Common Exception Types:**

## **Runtime Exceptions**

Runtime exceptions extend RuntimeException. They don’t have to be handled or declared. They can be thrown by the programmer or by the JVM. Common runtime exceptions include the following:

* **ArithmeticException** Thrown by the JVM when code attempts to divide by zero
* **ArrayIndexOutOfBoundsException** Thrown by the JVM when code uses an illegal index to access an array
* **ClassCastException** Thrown by the JVM when an attempt is made to cast an exception to a subclass of which it is not an instance
* **IllegalArgumentException** Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument
* **NullPointerException** Thrown by the JVM when there is a null reference where an object is required
* **NumberFormatException** Thrown by the programmer when an attempt is made to convert a string to a numeric type but the string doesn’t have an appropriate format
* **NumberFormatException** is type of **IllegalArgumentException** exception

## Checked Exceptions:

* Checked exceptions have Exception in their hierarchy but not RuntimeException. They must be handled or declared.
* **FileNotFoundException** Thrown programmatically when code tries to reference a file that does not exist
* **IOException** Thrown programmatically when there’s a problem reading or writing a file
* Also keep in mind that FileNotFoundException is a subclass of IOException

## Errors:

Errors extend the Error class. They are thrown by the JVM and should not be handled or declared. Errors are rare, but you might see these:

* **ExceptionInInitializerError** Thrown by the JVM when a static initializer throws an exception and doesn’t handle it
* **StackOverflowError** Thrown by the JVM when a method calls itself too many times (this is called infi nite recursion because the method typically calls itself without end)
* **NoClassDefFoundError** Thrown by the JVM when a class that the code uses is available at compile time but not runtime

# **Calling Methods That Throw Exceptions:**

# You might have noticed that eatCarrot() didn’t actually throw an exception; it just declared that it could. This is enough for the compiler to require the caller to handle or declare the exception.

# If you call a method which throws a checked exception you have two options to deal with that

# Surround that method with try,catch block Add the throws clause with the method in which you are calling the method.

# If you catch any **checked** exception which will be never thrown from try block will cause a compilation error. As the code in the catch block will be unreachable.

## Subclasses:

* When a class overrides a method from a supper class or implements a method from an interface it is not allowed to add any checked exceptions to that.
* A subclass is allowed to declare fewer exceptions than the superclass or interface. This is legal because callers are already handling them. A subclass not declaring an exception is similar to a method declaring it throws an exception that it never actually throws. This is perfectly legal.
* Similarly, a class is allowed to declare a subclass of an exception type. The idea is the same.
* This rule applies only to checked exceptions. The following code is legal because it has a runtime exception in the subclass’s version:

## Printing an Exception:

There are three ways to print an exception.

**public** **static** **void** main(String[] args) {

**try** {

hop();

} **catch** (Exception e) {

System.***out***.println(e);

System.***out***.println(e.getMessage());

e.printStackTrace();

}

}

**private** **static** **void** hop() {

**throw** **new** RuntimeException("cannot hop");

}

* First line will only print the exception type and the message.
* Second line print the message only.
* Third line print the complete stack trace of the exception