Chapter 10 : Exceptions

The basic philosophy of Java is that “badly formed code will not be run”.

Runtime : Allow the originator of the error to pass appropriate information to a recipient who will know how to handle the difficulty properly.

At the point where the exception occurs you might not know what to do with it. So you hand the problem out to a higher context.

Exception separates the code that describes what you want to do from the code that is executed when things go awry.

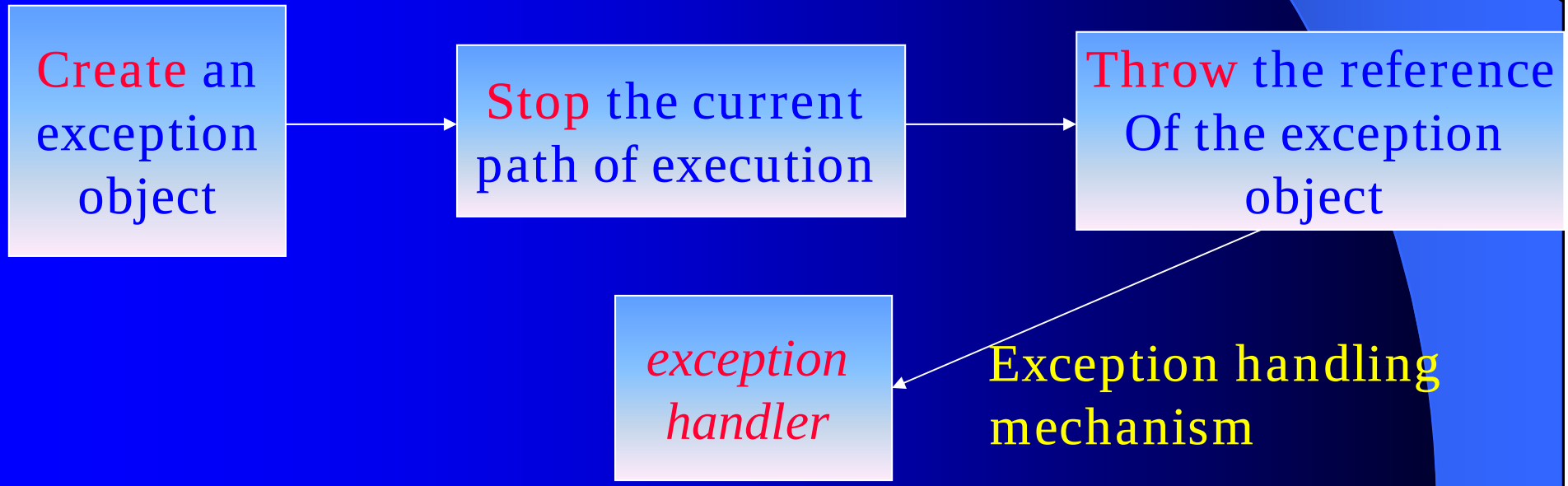
Exception handling is enforced by the Java compiler.

10.1 Basic exceptions

An *exceptional condition* is a problem that prevents the continuation of the method or scope.

a normal problem ⟷ an exceptional condition

throw an exception :



Exception arguments :

There are two constructors in all standard exceptions: the first is the default constructor, and the second takes a string argument.

You can throw any type of **Throwable** object. Typically, you'll throw a different class of exception for each different type of error.

The information about the error is represented both inside the exception object and implicitly in the type of exception object.

10.2 Catching an exception

Java exception handling allows you to concentrate on the problem in one place, and then deal with the errors from that code in another place.

Guarded region is a section of code that might produce exceptions, and is followed by the code to handle those exceptions.

The try block :

If you don't want a **throw** to exit a method, you can set up a special block within that method to capture the exception.

Exception handlers :

The thrown exception must end up someplace. ---- there's one handler for every exception type you want to catch.

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}
```

Each catch clause is like a little method that takes only one argument of a particular type.

If an exception is thrown, the exception handling mechanism goes hunting for the first handler with an matched argument. Then it enters that catch clause, and the exception is considered handled.

10.3 Creating your own exceptions

To create your own exception class, you're forced to inherit from an existing type of exception. The most trivial way is just to let the compiler create the default constructor.

Sample : SimpleExceptionDemo.java

The result is printed to the console *standard error* stream by writing to **System.err**. This is usually a better place to send error information than **System.out**.

Sample : FullConstructors.java *

stack trace :

```
Throwing MyException from f()
MyException
        at FullConstructors.f(FullConstructors.java:16)
        at FullConstructors.main(FullConstructors.java:24)
Throwing MyException from g()
MyException: Originated in g()
        at FullConstructors.g(FullConstructors.java:20)
        at FullConstructors.main(FullConstructors.java:29)
```

10.4 The exception specification

In Java, you're required to inform the client programmer of the exceptions that might be thrown from your method.

The exception specification uses an additional keyword, **throws**, followed by a list of all the potential exception types.

```
void f() throws TooBig, TooSmall, DivZero { //...
```

Note : If your method causes exceptions and doesn't handle them, the compiler will tell you that you must either handle the exception or write an exception specification.

You can claim to throw an exception that you really don't. It's also important for creating **abstract** base classes and **interfaces**.

10.5 Catching any exception

You do this by catching the base-class exception type **Exception.**

If you use it you should put it at the *end* of your list of handlers.

10.6 Rethrowing an exception

Rethrowing an exception causes the exception to go to the exception handlers in the next-higher context. Everything about the exception object is preserved.

If you want to install new stack trace information, you can do so by calling **fillInStackTrace( )**.

Sample : Rethrowing.java

Exceptions are all heap-based objects created with **new**, so the garbage collector automatically cleans them all up.

10.7 Standard Java exceptions

The name of the exception represents the problem that occurred, and the exception name is intended to be relatively self-explanatory.

RuntimeException :

They're always thrown automatically by Java and you don't need to include them in your exception specifications.

Because they indicate bugs, you virtually never catch a **RuntimeException**—it's dealt with automatically.

Sample : NeverCaught.java

If a **RuntimeException** gets all the way out to **main( )** without being caught, **printStackTrace( )** is called as the program exits.

10.8 Performing cleanup with finally

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

Whether an exception is thrown or not, the **finally** clause is always executed.

Sample : FinallyWorks.java

Exceptions in Java do not allow you to resume back to where the exception was thrown. However, If you place your **try** block in a loop, you can establish a condition that must be met before you continue the program.

10.8.1 What's finally for?

finally is necessary when you need to set something *other* than memory back to its original state.

Sample : WithFinally.java

Even the exception is not caught in the current set of **catch** clauses, **finally** will be executed before the exception handling mechanism continues its search for a handler at the next higher level.

Sample : AlwaysFinally.java

10.9 Exception restrictions

When you override a method, you can throw only the exceptions that have been specified in the base-class version of the method.

It means that code that works with the base class will automatically work with any object derived from the base class, including exceptions.

Sample : StormyInning.java *

The restriction on exceptions does not apply to constructors.

A derived-class version of a method may choose not to throw any exceptions, even if the base-class version does.

If you're dealing with exactly a **StormyInning** object, the compiler forces you to catch only the exceptions specific to that class, but if you upcast to the base type then the compiler forces you to catch the exceptions for the base type.

The "exception specification interface" for a particular method may narrow during inheritance and overriding, but it may not widen—the opposite of the rule for the class interface during inheritance.

10.10 Exception matching

When an exception is thrown, the exception handling system looks through the “nearest” handlers in the order they are written. When it finds a match, the exception is considered handled.

A derived-class object will match a handler for the base class.

Sample : Human.java