

Main Exception Classes

java.lang.Throwable

The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.

A throwable contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error. Finally, it can contain a cause: another throwable that caused this throwable to get thrown. The cause facility is new in release 1.4. It is also known as the chained exception facility, as the cause can, itself, have a cause, and so on, leading to a "chain" of exceptions, each caused by another

java.lang.Error

An Error indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

java.lang.Exception

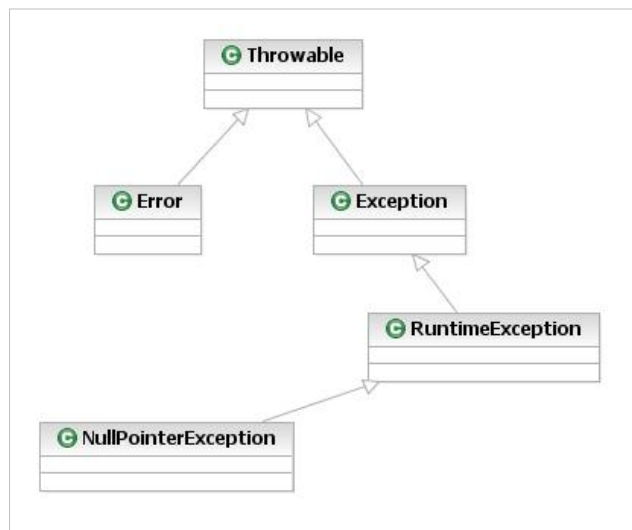
The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch. Also this is the class that a programmer may want to extend when adding business logic exceptions.

java.lang.RuntimeException

RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine. A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

java.lang.NullPointerException

Thrown when an application attempts to use null in a case where an object is required.



See Also

- [Java Programming/Keywords/try](#)
- [Java Programming/Keywords/catch](#)
- [Java Programming/Keywords/throws](#)
- [Java Programming/Keywords/throw](#)

Navigate **Exceptions** topic:

- Throwing and catching exceptions
 - Application exceptions
 - Runtime exceptions
 - NullPointerException
 - Main Exception classes
- Stack trace
- Checked exceptions

- Unchecked exceptions
- Preventing NullPointerException
- Chained exceptions(Nesting Exceptions)

This page describes some techniques for preventing NullPointerException.

It does not describe general techniques for how you should program Java. It is of some use, to make you more aware of null values, and to be more careful about generating them yourself.

Note that this list is not complete - there are no rules for preventing NullPointerException entirely in Java, because the standard libraries have to be used, and they can cause NullPointerExceptions. Also, it is possible to observe an uninitialised final field in Java, so you can't even treat a final field as being completely trusted during the object's creation.

A good approach is to learn how to deal with NullPointerExceptions first, and become competent with that. These suggestions will help you to cause less NullPointerExceptions, but they don't replace the need to know about NullPointerExceptions.

Minimize the use the keyword 'null' in assignment statements

This means not doing things like:

```
String s=null;
while (something)
    if (something2)
        s="yep";

if (s!=null)
    something3(s);
```

You can replace this with:

```
boolean done=false;

while (!done && something)
    if (something2)
    {
        done=true;
        something3("yep");
    }
```

You might also consider replacing null with "" in the first example, but default values bring about bugs caused by default values being left in place. A NullPointerException is actually better, as it allows the runtime to tell you about the bug, rather than just continue with a default value.

Minimize the use of the new Type[int] syntax for creating arrays of objects

An array created using `new Object[10]` has 10 null pointers. That's 10 more than we want, so use collections instead, or explicitly fill the array at initialisation with:

```
Object[] objects={"blah",5,new File("/usr/bin")};
```

or:

```
Object[] objects;  
objects=new Object[]{"blah",5,new File("/usr/bin")};
```

Check all references obtained from 'untrusted' methods

Many methods that can return a reference can return a null reference. Make sure you check these. For example:

```
File file=new File("/etc");  
File[] files=file.listFiles();  
if (files!=null)  
{  
    stuff  
}
```

`File.listFiles()` can return null if `"etc"` is not a directory.

You can decide to trust some methods not to return null, if you like, but that's an assumption you're making. Some methods that don't specify that they might return null, actually do, instead of throwing an exception.

Comparing string variable with a string literal

When you compare a variable with a string literal, always put the string literal first. For example do:

```
if ( "OK".equals( state ) )  
{  
    ...  
}
```

and do not do:

```
<strike>  
if ( state.equals( "OK" ) )  
{  
    ...  
}  
</strike>
```

If the `'state'` variable is null, you get a `NullPointerException` in the second example, but not in the first one.