

1.

A. "Base class's display() function has return type —— Object

Derived class's display() function has return type —— String

Just like every other class in Java, String class extends the Object class i.e. String is a sub-type of Object. Hence we can use it as return type in overridden display() function instead of type Object as in Base class."

source(["https://thispointer.com/overriding-a-method-with-different-return-type-in-java/"](https://thispointer.com/overriding-a-method-with-different-return-type-in-java/))

B. C# supports covariance

```
C# Copy

// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

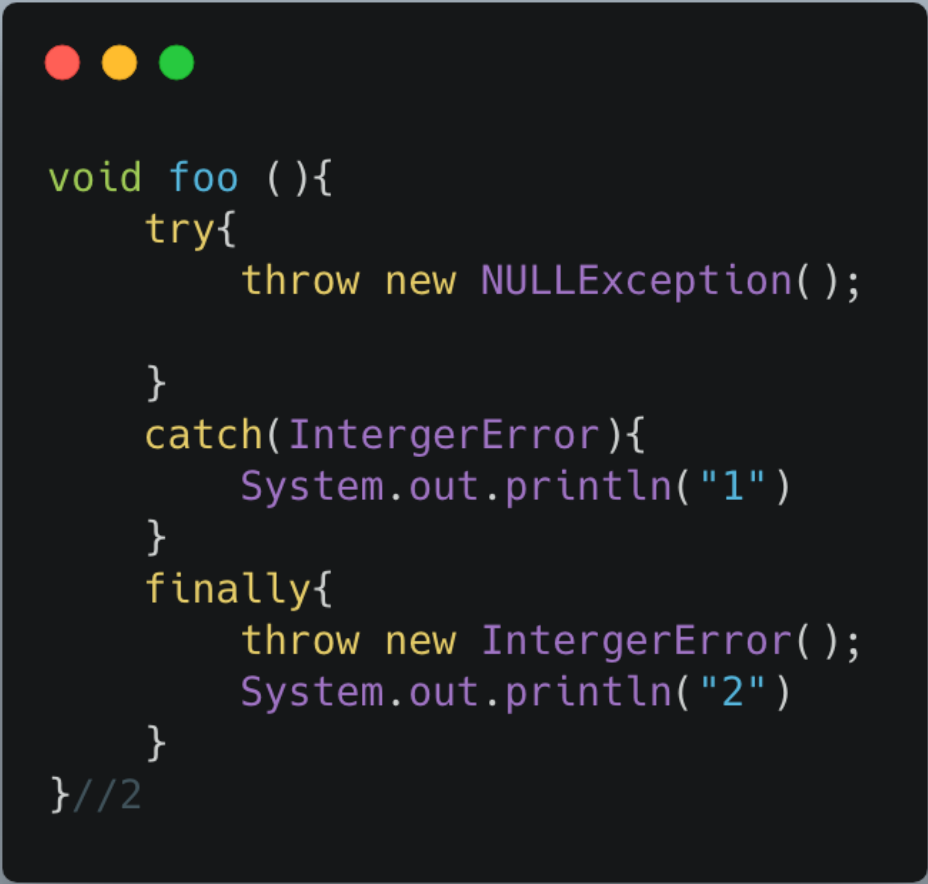
It makes it easier to maintain code in the long term and does not require a separate array variable for each type of subtype.

Source(MS Docs)

C C# also support contravariance


Reasoning above

This rule can help make designing compare functions easy such that you are able to take in two objects and produce some sort of information relating to compare.



```
void foo () {  
    try {  
        throw new NullPointerException();  
    }  
    catch (IntergerError) {  
        System.out.println("1")  
    }  
    finally {  
        throw new IntergerError();  
        System.out.println("2")  
    }  
} //2
```


The try block cannot handle the `NullPointerException` so it goes into the finally where another exception is raised. The program outputs 2



```
void foo(){  
    try{  
        throw new NullPointerException();  
    }  
    catch(None){  
        System.out.println("1")  
    }  
    finally{  
        System.out.println("return")  
        return;  
    }  
} //return
```

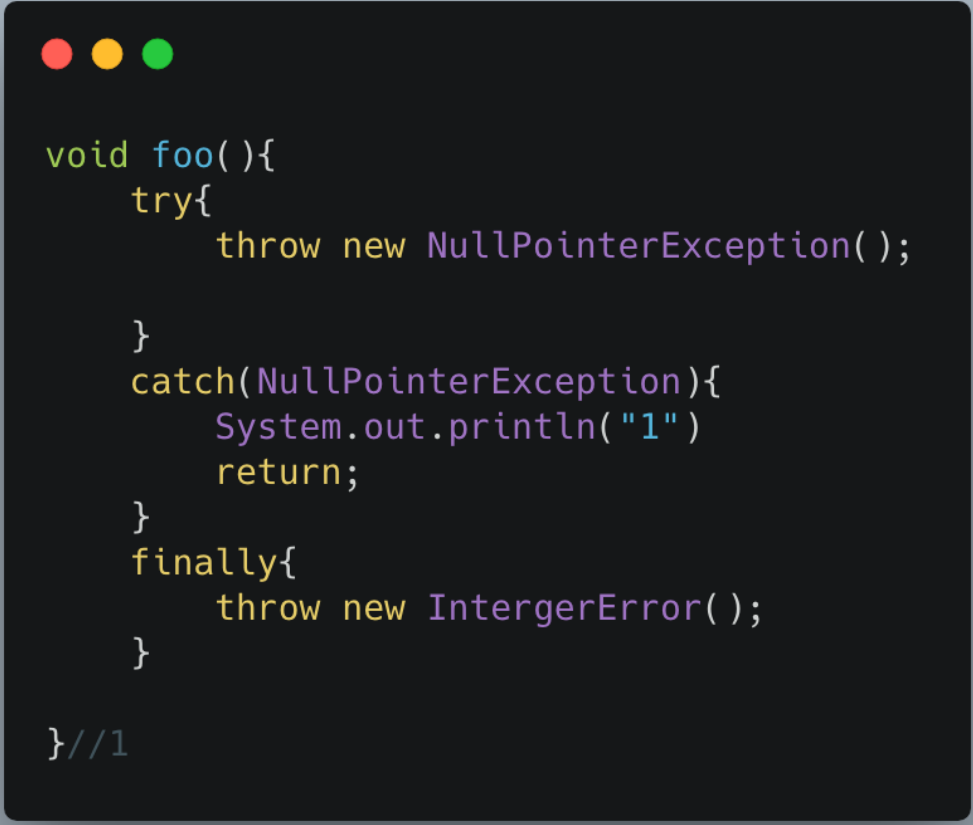
output: return

The catch block does not catch the `NullPointerException` so finally returns with an error.



```
void foo(){
    try{
        throw new NullPointerException();
    }
    catch(NullPointerException){
        System.out.println("1")
    }
    finally{
        System.out.println("2")
        return
    }
} //1 2
```

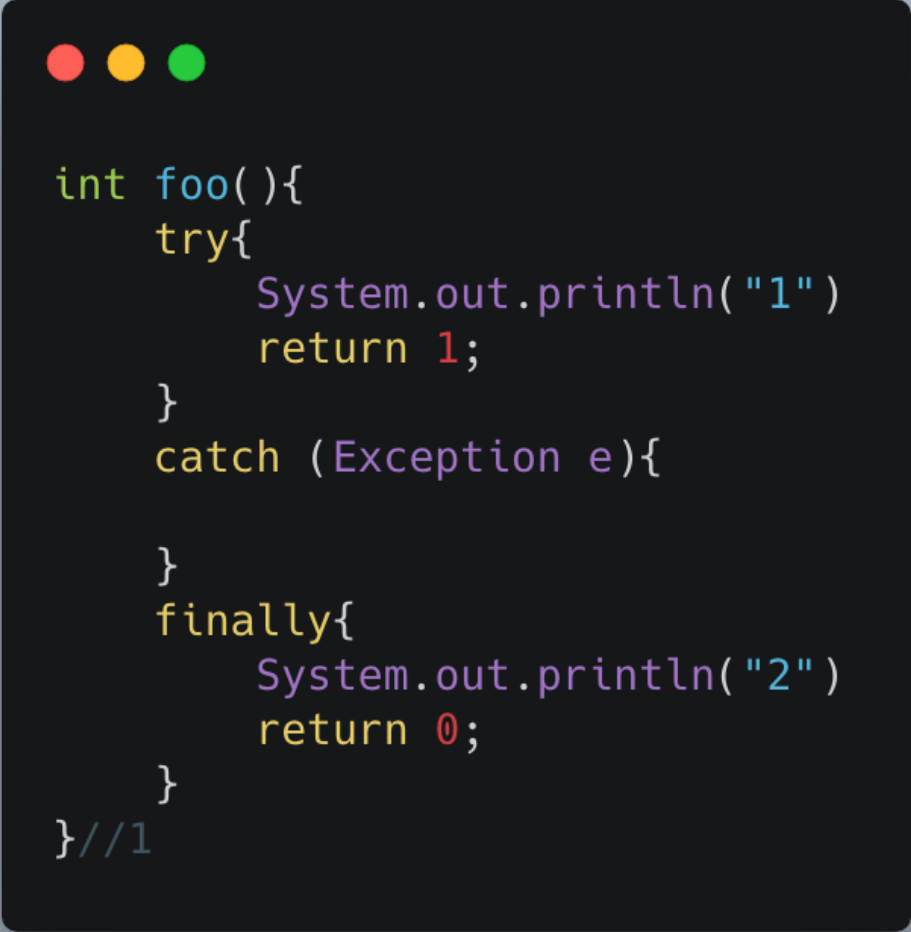
The exception thrown will be caught by the catch block and finally will execute.
output: 1 2



```
void foo(){  
    try{  
        throw new NullPointerException();  
    }  
    catch(NullPointerException){  
        System.out.println("1")  
        return;  
    }  
    finally{  
        throw new IntergerError();  
    }  
}
```

//1

The NullPointerException is caught and the function is return. The finally block never executes
output: 1



```
int foo(){  
    try{  
        System.out.println("1")  
        return 1;  
    }  
    catch (Exception e){  
  
    }  
    finally{  
        System.out.println("2")  
        return 0;  
    }  
} //1
```

The code ends at the try statement and the finally block is never executed
output 1

A 0,2

B 3,2

C 3,2

D 1,3

E 1,3

Ruby supports pass by value

```
def test(a1,a2)
  a1 = "C"
  a2= "C++"

end
#test "C", "C++"
a = "C"
b = "C"
test(a,b)
puts a
puts b
#C |
```

The value of a1 in test is C and a2 = C++ in test function

```
x = '10'

def change_value(val)
  val << '20'
end

puts x # 10
change_value(x)
puts x # 1020
```

Source :<http://rubyblog.pro/2017/09/pass-by-value-or-pass-by-reference>