



Exceptions

Handling exceptional situations

What can possibly go wrong?

Goal: call ratio

```
class Wrong {  
    public int ratio(Integer a, Integer b) {  
        return a/b;  
    }  
}
```

Exceptions

- Exceptional Situations
- Syntax of **try**, **catch**, **throw**, **finally**, and **throws**
- Defining and using your own exceptions

Exceptions are for **exceptional** situations

When atypical situations happen:

- errors in computation or data,
- invalid parameters,
- failure to complete tasks

- Exceptions indicate the problems (e.g., `NullPointerException` reflects problems with referencing `null`.)

- Some, but not all, should be handled.
- Handle exceptions by *catching* them.
- It is possible to delegate an exception to the caller by *throwing* it.

Exception Handling Syntax

Until now, you have had no control over exceptions. With a **catch** statement, you can implement your own exception handling.

```
try {  
    // Code "in question"  
}  
catch (exception_type name) {  
    // Code in response to exception  
}  
finally {  
    // Code guaranteed to be  
    // executed after try  
    // (and previous catches)  
}
```

Idea:

- try some code,
- catch exception from tried code,
- Optional: finally, "clean up" if necessary

Exception Handling Example #1

```
public class TryCatch1 {  
    public static void main(String args[]) {  
        int nums[] = new int[3];
```

```
try {
```

Create a try block

```
    System.out.println("Before offending line");  
    // attempt to run off an array  
    nums[10] = 28;  
    System.out.println("Will this be displayed?");  
}
```

Catch this exception or subclass

```
catch (ArrayIndexOutOfBoundsException e){  
    // catch an exception as e  
    System.out.println("Index out of bounds!");  
}  
System.out.println("Done");
```

```
}  
}
```

Exception Handling Example #2

Idea: Uncaught exceptions propagate to the caller and eventually caught by the JVM, terminating the program.

```
public class TryCatch2 {  
    static void fooBar() {  
        int nums[] = new int[3];  
        System.out.println("Before offending line");  
        nums[10] = 28;  
        System.out.println("After the offending line.");  
    }  
}
```

```
public static void main(String args[]) {  
    try {  
        fooBar();  
    }  
    catch (ArrayIndexOutOfBoundsException e) {  
        // catch an exception  
        System.out.println("Index out of bounds!");  
    }  
    System.out.println("Done");  
}
```

Exception Handling Example #3

```
public class TryCatch3 {  
    public static void main(String args[]) {  
        int numbers[] = {4, 2, 7, 9};    // length: 4  
        int denoms[] = {2, 5, 0};        // length: 3  
  
        for (int i=0;i<numbers.length;i++) {  
            try {  
                System.out.printf("%d/%d = %d\n",  
                    numbers[i], denoms[i], numbers[i]/denoms[i]);  
            }  
            catch (ArithmeticException e) {  
                System.out.println("Can't divide by zero");  
            }  
            catch (ArrayIndexOutOfBoundsException e) {  
                System.out.println("No matching element");  
            }  
        }  
        System.out.println("DONE");  
    }  
}
```

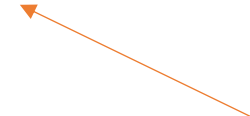
Idea: Multiple catches possible.
Considered serially just like if's.

Throwing an Exception

- So far: catch exceptions someone else generated
- Can create or pass on an exception as well. It is called **throwing** an exception:

throw *{an exception object}*

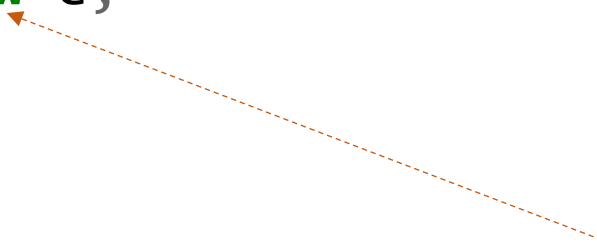
```
int offerMod(int x, int y) {  
    if (y == 0)  
        throw new ArithmeticException();  
    return x % y;  
}
```



Exception happens on this line

Catching and Rethrowing

```
void foo() {  
    int numbers[] = {4, 2, 7, 9};    // Length: 4  
    int denoms[] = {2, 5, 0};        // Length: 3  
  
    for (int i=0;i<numbers.length;i++) {  
        try { System.out.printf("%d/%d = %d\n",  
                                numbers[i], denoms[i], numbers[i]/denoms[i]);  
        } catch (ArithmeticException e) {  
            System.out.println("Can't divide by zero");  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("No matching element");  
            throw e;  
        }  
    }  
}
```

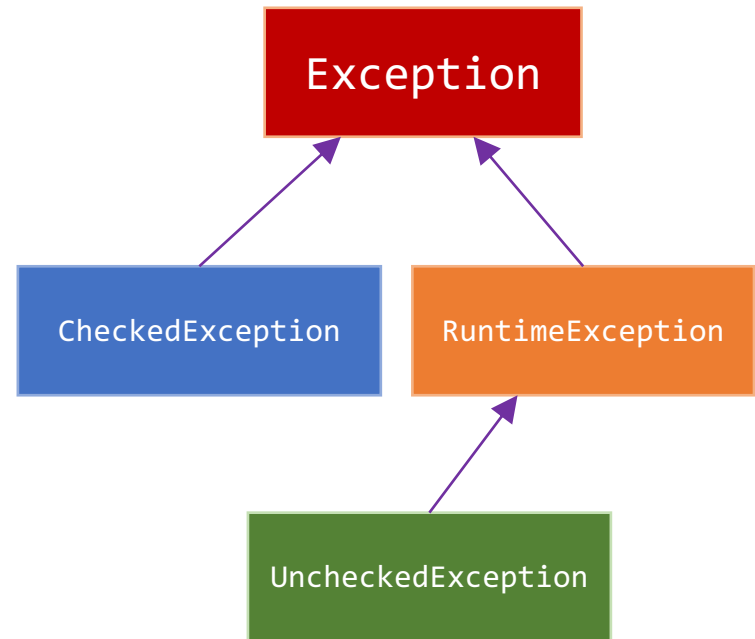


Idea: An exception can be caught, handled, and passed on (known as **re-throw**) to the outer caller.

Deep Dive: Exceptions in Java

An *exception* is a subclass of **Exception**.
Two flavors:

- **Checked Exceptions:**
those that must be explicitly “treated” somehow, such as **IOException** – e.g., an issue reading a file
- **Unchecked Exceptions:**
those that do not; e.g., **RuntimeException** such as **NullPointerException**



Java's Built-in Exceptions

Exception	Meaning
ArithmeticException	Arithmetic error, such as integer divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	
EnumConstantNotPresentException	
IllegalArgumentException	
IllegalMonitorStateException	
IllegalStateException	
IllegalThreadStateException	
IndexOutOfBoundsException	
NegativeArraySizeException	
NullPointerException	
NumberFormatException	
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Define Your Own Exceptions

- You can define and throw your own specialized exceptions (e.g., **DataOutOfBoundsException**, **QueueEmptyException**).
- Useful for responding to situations not covered by Java's predefined exceptions.

```
public class DataOutOfBoundsException extends Exception {  
    public DataOutOfBoundsException(String dataName){  
        super("Data value " + dataName + " is out of bounds.");  
    }  
}
```

Q: Is this checked or unchecked?

Using Your Exception

Idea: *Checked Exception*. Every method that can come out with a “checked” exception must declare what exceptions it may throw in the method declaration.

```
class Person {  
    private int age;  
    private String name;  
    /* ... */  
}
```

```
public void setAge(int age) throws DataOutOfBoundsException {  
    if (age < 0 || age > 120) {  
        throw new DataOutOfBoundsException(""+age);  
    }  
    this.age = age;  
}
```

Example: **setAge** is potentially throwing a checked exception and is declared as such.

Catching Your Custom Exception

Catching it is similar to catching any other exception.

```
public void makePerson() {  
    Person clone = new Person();  
    int random = (int)(Math.random() * 1000);  
    try {  
        clone.setAge(random);  
    }  
    catch (DataOutOfBoundsException e) {  
        System.out.println(e.getMessage());  
        // gets the message describing the problem  
    }  
}
```

Exceptions: Pros and Cons

Pros:

- cleaner code: rather than returning an error up chain of calls to check for exceptional cases, throw an exception!
- So that we use return value for meaningful data, not error checking
- factor out error-checking code into one class, so it can be reused.

Cons:

- throwing exceptions isn't free (requires computation)
- can become messy if not used sparingly
- can cover up serious problems, if not careful. For example, catching a **NullPointerException** and be silent about it.

Words of Wisdom:

- Never try to “cover up” your program by catching all exceptions.
- Best to throw an exception when an error occurs that you cannot deal with yourself, but can be better handled by some method further up the chain.