Software Engineering 2 (C++)

CSY2006 (Week 18)

Dr. Suraj Ajit

Exceptions

- Indicate that something unexpected has occurred or been detected
- Allow program to deal with the problem in a controlled manner
- Can be as simple or complex as program design requires

Exceptions - Terminology

- Exception: object or value that signals an error
- Throw an exception: send a signal that an error has occurred
- Catch/Handle an exception: process the exception; interpret the signal

Exceptions – Key Words

- `throw` – followed by an argument, is used to throw an exception
- `try` – followed by a block `{ }`, is used to invoke code that throws an exception
- `catch` – followed by a block `{ }`, is used to detect and process exceptions thrown in preceding `try` block. Takes a parameter that matches the type thrown.

Exceptions – Flow of Control

1) A function that throws an exception is called from within a try block
2) If the function throws an exception, the function terminates and the try block is immediately exited. A catch block to process the exception is searched for in the source code immediately following the try block.
3) If a catch block is found that matches the exception thrown, it is executed. If no catch block that matches the exception is found, the program terminates.

Exceptions – Example (1)

```
// function that throws an exception
int totalDays(int days, int weeks)
{
   if ((days < 0) || (days > 7))
     throw "invalid number of days";
// the argument to throw is the
// character string
  else
     return (7 * weeks + days);
}
```

Exceptions – Example (2)

```
try // block that calls function
{
     totDays = totalDays(days, weeks);
    cout << "Total days: " << days;
}
catch (char *msg) // interpret
                                  //  exception
{
    cout << "Error: " << msg;
}
```

Exceptions – What Happens

1) `try` block is entered. `totalDays` function is called
2) If 1st parameter is between 0 and 7, total number of days is returned and `catch` block is skipped over (no exception thrown)
3) If exception is thrown, function and `try` block are exited, `catch` blocks are scanned for 1st one that matches the data type of the thrown exception. `catch` block executes

From Program 16-1

```
 8  int main()
 9  {
10     int num1, num2;  // To hold two numbers
11     double quotient; // To hold the quotient of the numbers
12
13     // Get two numbers.
14     cout << "Enter two numbers: ";
15     cin >> num1 >> num2;
16
17     // Divide num1 by num2 and catch any
18     // potential exceptions.
19     try
20     {
21        quotient = divide(num1, num2);
22        cout << "The quotient is " << quotient << endl;
23     }
24     catch (char *exceptionString)
25     {
26        cout << exceptionString;
27     }
28
29     cout << "End of the program.\n";
30     return 0;
31  }
```

From Program 16-1

```
33  //*********************************************
34  // The divide function divides numerator by  *
35  // denominator. If denominator is zero, the  *
36  // function throws an exception.             *
37  //*********************************************
38
39  double divide(int numerator, int denominator)
40  {
41     if (denominator == 0)
42        throw "ERROR: Cannot divide by zero.\n";
43
44     return static_cast<double>(numerator) / denominator;
45  }
```

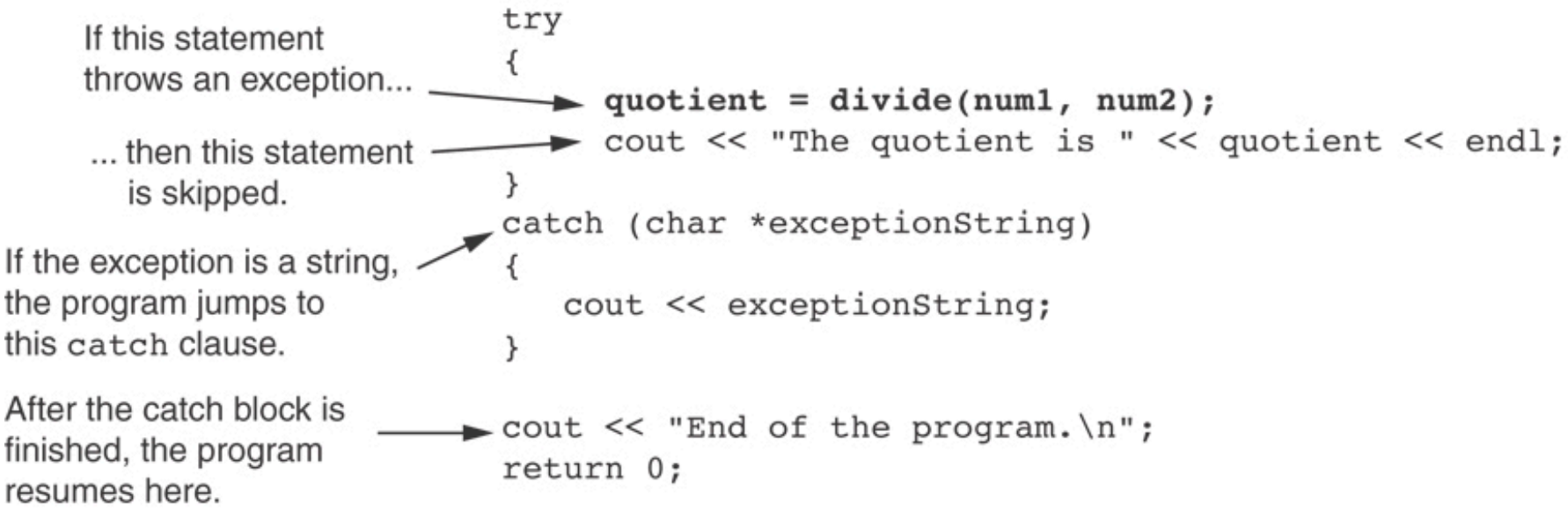
Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 2 [Enter]
The quotient is 6
End of the program.
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 0 [Enter]
ERROR: Cannot divide by zero.
End of the program.
```

What Happens in theTry/Catch Construct



What if no exception is thrown?

If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

```
try
{
      quotient = divide(num1, num2);
      cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

Exceptions - Notes

- Predefined functions such as `new` may throw exceptions
- The value that is thrown does not need to be used in `catch` block.
  - in this case, no name is needed in catch parameter definition
  - `catch` block parameter definition *does* need the type of exception being caught

Exception Not Caught?

- An exception will not be caught if
  - it is thrown from outside of a `try` block
  - there is no `catch` block that matches the data type of the thrown exception
- If an exception is not caught, the program will terminate

Exceptions and Objects

- An exception class can be defined in a class and thrown as an exception by a member function
- An exception class may have:
  - no members: used only to signal an error
  - members: pass error data to `catch` block
- A class can have more than one exception class

Contents of `Rectangle.h` (Version 1)

```
1  // Specification file for the Rectangle class
2  #ifndef RECTANGLE_H
3  #define RECTANGLE_H
4
5  class Rectangle
6  {
7     private:
8        double width;     // The rectangle's width
9        double length;    // The rectangle's length
10    public:
11       // Exception class
12       class NegativeSize
13          { };            // Empty class declaration
14
15       // Default constructor
16       Rectangle()
17          { width = 0.0; length = 0.0; }
18
19       // Mutator functions, defined in Rectangle.cpp
20       void setWidth(double);
21       void setLength(double);
22
```

Contents of Rectangle.h (Version1) (Continued)

```
23          // Accessor functions
24          double getWidth() const
25             { return width; }
26
27          double getLength() const
28             { return length; }
29
30          double getArea() const
31             { return width * length; }
32    };
33    #endif
```

Contents of `Rectangle.cpp` (Version 1)

```
1  // Implementation file for the Rectangle class.
2  #include "Rectangle.h"
3
4  //*************************************************************
5  // setWidth sets the value of the member variable width.    *
6  //*************************************************************
7
8  void Rectangle::setWidth(double w)
9  {
10    if (w >= 0)
11       width = w;
12    else
13       throw NegativeSize();
14 }
15
16 //*************************************************************
17 // setLength sets the value of the member variable length.  *
18 //*************************************************************
19
20 void Rectangle::setLength(double len)
21 {
22    if (len >= 0)
23       length = len;
24    else
25       throw NegativeSize();
26 }
```

Program 16-2

```
1  // This program demonstrates Rectangle class exceptions.
2  #include <iostream>
3  #include "Rectangle.h"
4  using namespace std;
5
6  int main()
7  {
8     int width;
9     int length;
10
11    // Create a Rectangle object.
12    Rectangle myRectangle;
13
```

Program 16-2 *(continued)*

```
14      // Get the width and length.
15      cout << "Enter the rectangle's width: ";
16      cin >> width;
17      cout << "Enter the rectangle's length: ";
18      cin >> length;
19
20      // Store these values in the Rectangle object.
21      try
22      {
23         myRectangle.setWidth(width);
24         myRectangle.setLength(length);
25         cout << "The area of the rectangle is "
26              << myRectangle.getArea() << endl;
27      }
28      catch (Rectangle::NegativeSize)
29      {
30         cout << "Error: A negative value was entered.\n";
31      }
32      cout << "End of the program.\n";
33
34      return 0;
35  }
```

Program 16-2 (Continued)

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: 10 [Enter]
Enter the rectangle's length: 20 [Enter]
The area of the rectangle is 200
End of the program.
```

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: 5 [Enter]
Enter the rectangle's length: -5 [Enter]
Error: A negative value was entered.
End of the program.
```

See: Pr 16-3: Multiple Exceptions (separate exception class for negative length, negative width)
Pr 16-4: Better Exception Handling
Pr 16-5: Passing parameter to Exception class

What Happens After catch Block?

- Once an exception is thrown, the program cannot return to throw point. The function executing `throw` terminates (does not return), other calling functions in `try` block terminate, resulting in unwinding the stack
- If objects were created in the `try` block and an exception is thrown, they are destroyed.

Nested try Blocks

- `try/catch` blocks can occur within an enclosing `try` block
- Exceptions caught at an inner level can be passed up to a `catch` block at an outer level:

```
catch ( )
{
   ...
   throw;  // pass exception up
}          // to next level
```

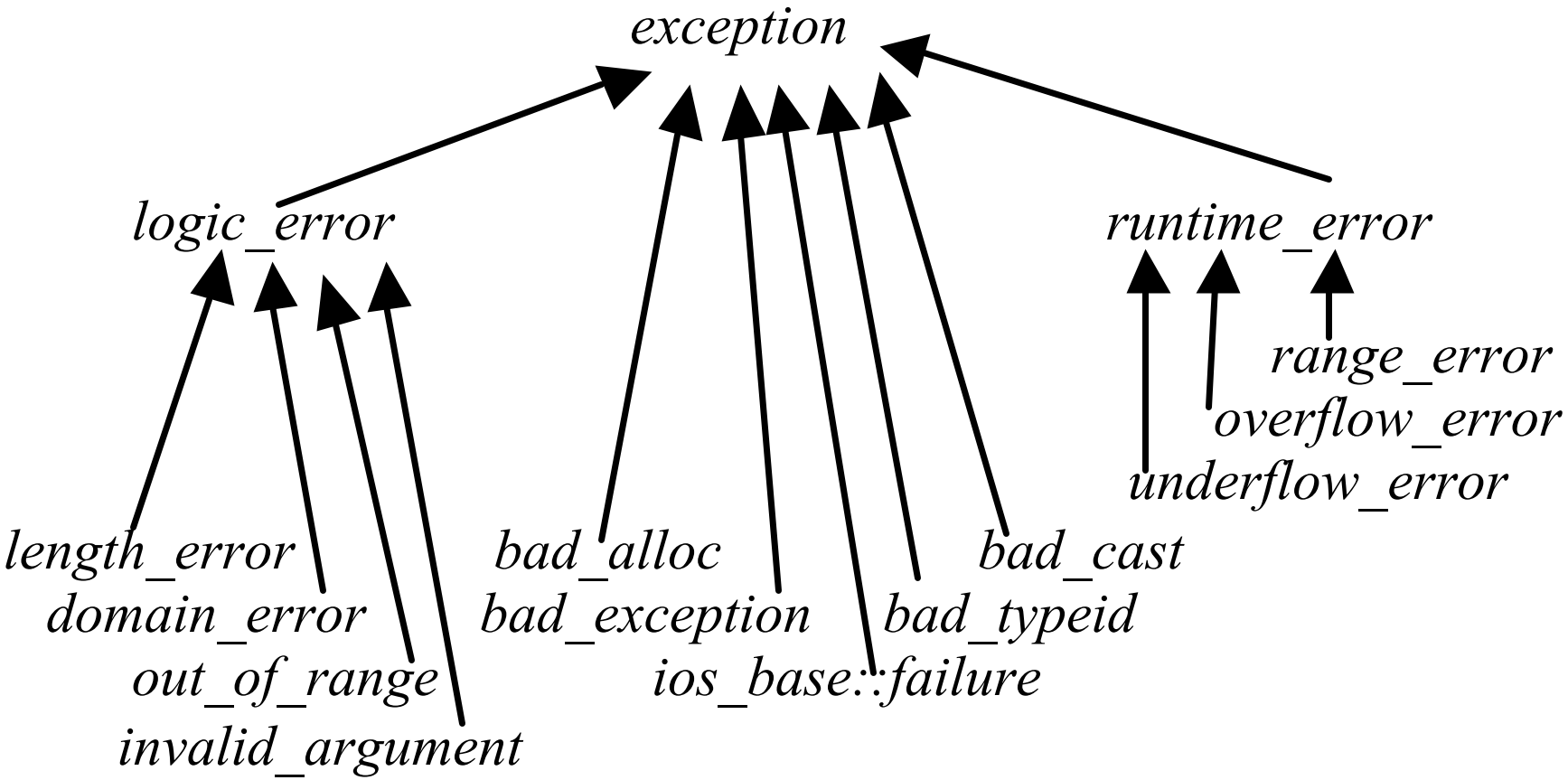
Testing Available Memory

- new operator throws bad_alloc exception if insufficient memory:

```
try
{
        NodePtr pointer = new Node;
}
catch (bad_alloc)
{
        cout << "Ran out of memory!";
        // Can do other things here as well…
}
```

- In library <new>, std namespace
- See: Pr 16-6

Exceptions Hierarchy



Further Reading/Reference

- See Extra Resources Folder (NILE)
  - Exception Notes
  - Notes and Examples from comparison of Java versus C++

Function Templates

Function Templates

- Function template: a pattern for a function that can work with many data types
- When written, parameters are left for the data types
- When called, compiler generates code for specific data types in function call

Examples: Pr 16-7 to Pr 16-10

Function Template Example

```
template <class T>
T times10(T num)
{
        return 10 * num;
}
```

template prefix

generic data type

type parameter

What gets generated when `times10` is called with an `int`:	What gets generated when `times10` is called with a `double`:

`int times10(int num)`
`{`
`  return 10 * num;`
`}`
	`double times10(double num)`
	`{`
	`  return 10 * num;`
	`}`

Function Template Example

```
template <class T>
T times10(T num)
{
        return 10 * num;
}
```

- Call a template function in the usual manner:

```
int ival = 3;
double dval = 2.55;
cout << times10(ival); // displays 30
cout << times10(dval); // displays 25.5
```

Function Template Notes

- Can define a template to use multiple data types:

```
template<class T1, class T2>
```

- Example:

```
template<class T1, class T2>      // T1 and T2 will be
double mpg(T1 miles, T2 gallons) // replaced in the
{                                 // called function
return miles / gallons          // with the data
}                                 // types of the
                                  // arguments
```

Function Template Notes

- Function templates can be overloaded Each template must have a unique parameter list

```
template <class T>
T sumAll(T num) ...
template <class T1, class T2>
T1 sumall(T1 num1, T2 num2) ...
```

Function Template Notes

- All data types specified in template prefix must be used in template definition
- Function calls must pass parameters for all data types specified in the template prefix
- Like regular functions, function templates must be defined before being called

Function Template Notes

- A function template is a pattern
- No actual code is generated until the function named in the template is called
- A function template uses no memory
- When passing a class object to a function template, ensure that all operators in the template are defined or overloaded in the class definition

Where to Start When Defining Templates

Where to Start When Defining Templates

- Templates are often appropriate for multiple functions that perform the same task with different parameter data types
- Develop function using usual data types first, then convert to a template:
  - add template prefix
  - convert data type names in the function to a type parameter (*i.e.*, a T type) in the template

Class Templates

Class Templates

- Classes can also be represented by templates. When a class object is created, type information is supplied to define the type of data members of the class.
- Unlike functions, classes are instantiated by supplying the type name (`int`, `double`, `string`, etc.) at object definition

Class Template Example

```
template <class T>
class grade
{
    private:
        T score;
    public:
        grade(T);
        void setGrade(T);
        T getGrade()
};
```

Class Template Example

- Pass type information to class template when defining objects:

```
grade<int> testList[20];

grade<double> quizList[20];
```

- Use as ordinary objects once defined

Class Templates and Inheritance

- Class templates can inherit from other class templates:

```
template <class T>
class Rectangle
  { ... };
template <class T>
class Square : public Rectangle<T>
  { ... };
```

- Must use type parameter `T` everywhere base class name is used in derived class

More examples: Pr 16-11 and Pr 16-12 use user-defined class templates

Introduction to the Standard Template Library

Introduction to the Standard Template Library

- Standard Template Library (STL): a library containing templates for frequently used data structures and algorithms
- Not supported by many older compilers

See examples:

16-11 and 16-12 are implementations of user-defined class templates

16-13 to 16-19 are examples of uses of pre-defined templates (STL Library)

Standard Template Library

- Two important types of data structures in the STL:
  - containers: classes that stores data and imposes some organization on it
  - iterators: like pointers; mechanisms for accessing elements in a container

Containers

- Two types of container classes in STL:
  - sequence containers: organize and access data sequentially, as in an array. These include `vector`, `dequeue`, and `list`
  - associative containers: use keys to allow data elements to be quickly accessed. These include `set`, `multiset`, `map`, and `multimap`

Iterators

- Generalization of pointers, used to access information in containers
- Four types:
  - forward (uses `++`)
  - bidirectional (uses `++` and `--`)
  - random-access
  - input (can be used with `cin` and `istream` objects)
  - output (can be used with `cout` and `ostream` objects)

Algorithms

- STL contains algorithms implemented as function templates to perform operations on containers.
- Requires `algorithm` header file
- `algorithm` includes

`binary_search`	`count`
`for_each`	`find`
`find_if`	`max_element`
`min_element`	`random_shuffle`
`sort`	and others

Introduction to the STL vector

Introduction to the STL vector

- A data type defined in the Standard Template Library
- Can hold values of any type:

```
vector<int> scores;
```

- Automatically adds space as more is needed – no need to determine size at definition
- Can use `[]` to access elements

See: examples Pr7-21 to Pr7-26

Declaring Vectors

- You must `#include<vector>`
- Declare a vector to hold `int` element:

```
vector<int> scores;
```

- Declare a vector with initial size 30:

```
vector<int> scores(30);
```

- Declare a vector and initialize all elements to 0:

```
vector<int> scores(30, 0);
```

- Declare a vector initialized to size and contents of another vector:

```
vector<int> finals(scores);
```

Adding Elements to a Vector

- Use `push_back` member function to add element to a full array or to an array that had no defined size:

```
scores.push_back(75);
```

- Use `size` member function to determine size of a vector:

```
howbig = scores.size();
```

Removing Vector Elements

- Use `pop_back` member function to remove last element from vector:

```
scores.pop_back();
```

- To remove all contents of vector, use `clear` member function:

```
scores.clear();
```

- To determine if vector is empty, use `empty` member function:

```
while (!scores.empty()) ...
```

Other Useful Member Functions

Member Function	Description	Example
`at(elt)`	Returns the value of the element at position `elt` in the vector	`cout << vec1.at(i);`
`capacity()`	Returns the maximum number of elements a vector can store without allocating more memory	`maxelts = vec1.capacity();`
`reverse()`	Reverse the order of the elements in a vector	`vec1.reverse();`
`resize (elts,val)`	Add elements to a vector, optionally initializes them	`vec1.resize(5,0);`
`swap(vec2)`	Exchange the contents of two vectors	`vec1.swap(vec2);`

Miscellaneous (Notes)

Exception Specification

- Functions that don’t catch exceptions
  - Should "warn" users that it could throw
  - But it won’t catch!
- Should list such exceptions:
  double safeDivide(int top, int bottom)
  throw (DividebyZero);
  - Called "exception specification" or "throw list"
  - Should be in declaration and definition
  - All types listed handled "normally"
  - If no throw list → all types considered there

Throw List

- If exception thrown in function NOT in throw list:
  - No errors (compile or run-time)
  - Function unexpected() automatically called
    - Default behavior is to terminate
    - Can modify behavior
- Same result if no catch-block found

Throw List Summary

- void someFunction()
  throw(DividebyZero, OtherException);
  //Exception types DividebyZero or OtherException
  //treated normally. All others invoke unexpected()
- void someFunction() throw ();
  //Empty exception list, all exceptions invoke unexpected()
- void someFunction();
  //All exceptions of all types treated normally

Derived Classes

- Remember: derived class objects also objects of base class
- Consider:
  D is derived class of B
- If B is in exception specification →
  - Class D thrown objects will also be treated normally, since it's also object of class B
- Note: does not do automatic type cast:
  - double will not account for throwing an int

unexpected()

- Default action: terminates program
  - No special includes or using directives
- Normally no need to redefine
- But you can:
  - Use set_unexpected
  - Consult compiler manual or advanced text for details

When to Throw Exceptions

- Typical to separate throws and catches
  - In separate functions
- Throwing function:
  - Include throw statements in definition
  - List exceptions in throw list
    - In both declaration and definition
- Catching function:
  - Different function, perhaps even in different file

Preferred throw-catch Triad: throw

- void functionA() throw (MyException)
  {
  …
  throw MyException(arg);
  …
  }

- Function throws exception as needed

Preferred throw-catch Triad: catch

- Then some other function:

```
void functionB()
{
        …
        try
        {
                …
                functionA();
                …
        }
        catch (MyException e)
        { // Handle exception
        }
        …
}
```

Uncaught Exceptions

- Should catch every exception thrown
- If not → program terminates
  - terminate() is called
- Recall for functions
  - If exception not in throw list: unexpected() is called
    - It in turn calls terminate()
- So same result

Overuse of Exceptions

- Exceptions alter flow of control
  - Similar to old "goto" construct
  - "Unrestricted" flow of control
- Should be used sparingly
- Good rule:
  - If desire a "throw": consider how to write program without throw
  - If alternative reasonable → do it

Exception Class Hierarchies

- Useful to have; consider:
  DivideByZero class derives from:
  ArithmeticError exception class
  - All catch-blocks for ArithmeticError also catch DivideByZero
  - If ArithmeticError in throw list, then DividebyZero also considered there

Testing Available Memory

- new operator throws bad_alloc exception if insufficient memory:

```
try
{
        NodePtr pointer = new Node;
}
catch (bad_alloc)
{
        cout << "Ran out of memory!";
        // Can do other things here as well…
}
```

- In library <new>, std namespace

Rethrowing an Exception

- Legal to throw exception IN catch-block!
  - Typically only in rare cases
- Throws to catch-block "farther up chain"
- Can re-throw same or new exception
  - rethrow;
    - Throws same exception again
  - throw newExceptionUp;
    - Throws new exception to next catch-block

Summary 1

- Exception handling allows separation of "normal" cases and "exceptional" cases
- Exceptions thrown in try-block
  - Or within a function whose call is in try-block
- Exceptions caught in catch-block
- try-blocks typically followed by more than one catch-block
  - List more specific exceptions first

Summary 2

- Best used with separate functions
  - Especially considering callers might handle differently
- Exceptions thrown in but not caught in function, should be listed in throw list
- Exceptions thrown but never caught → program terminates
- Resist overuse of exceptions
  - Unrestricted flow of control