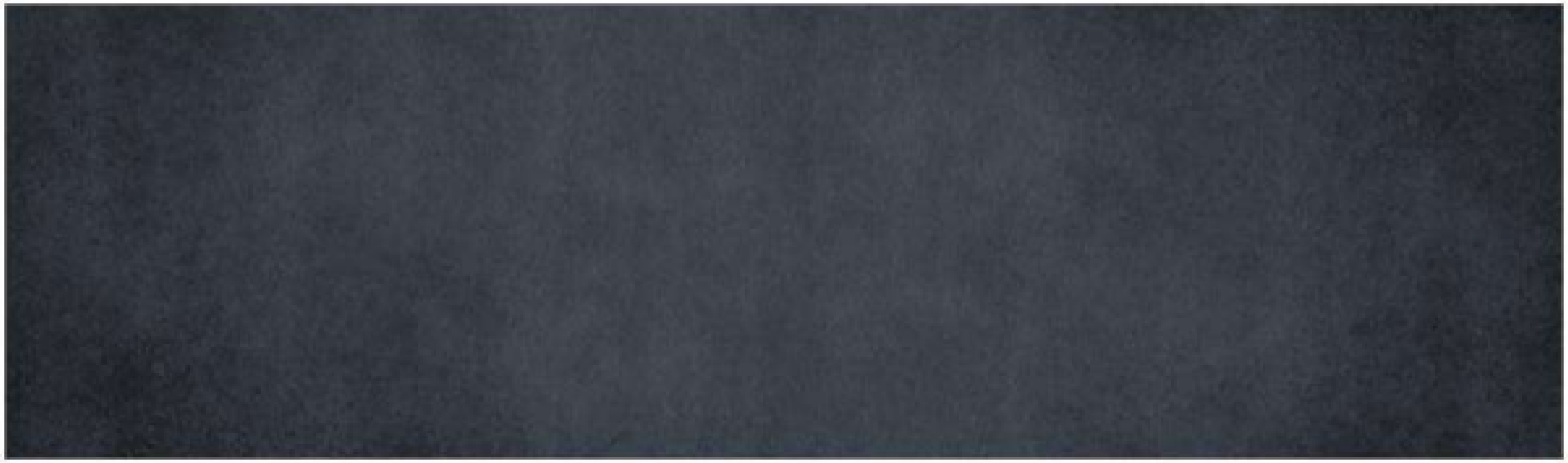
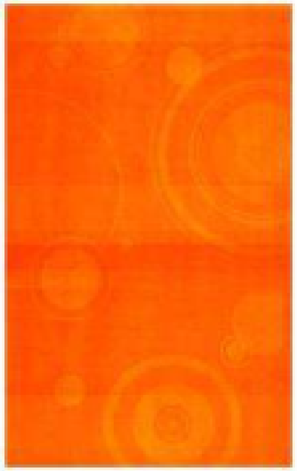



What is an Exception?

An exception is an unexpected event that occurs during runtime and causes normal program flow to be disrupted.

Some common examples:

▪ Divide by zero errors
▪ Accessing the elements of an array beyond its range ▪ Invalid input
▪ Hard disk crash
▪ Opening a non-existent file
▪ Heap memory exhaustion

Exception Handling

The way of handling anomalous

situations in a program-run is known as Exception Handling.

Its advantages are:

▪ Exception handling separates error-handling code from normal code

▪ It clarifies the code and enhances readability

▪ It stimulates consequences as the error-handling takes place at one place and in one manner

▪ It makes for clear, robust and fault-tolerant programs

1

- Write code such that it raises an error flag every time something goes wrong

- [throw exception]

- [catch exception]

Concept of Exception Handling

Terminology

Checked Exception

Checked exceptions are inherited from the core Java class Exception.
They represent exceptions that are frequently considered “non fatal” to program execution

Checked exceptions must be handled in your code, or passed to parent classes for handling

Unchecked Exception

Unchecked exceptions represent error conditions that are considered "fatal" to program execution.
You do not have to do anything with an unchecked exception. Your program will terminate with an appropriate error message

If the method contains code that may cause a checked exception, we <u>MUST</u> handle the exception <u>OR</u> pass the exception to the parent class (every class has Object as the ultimate parent)

₪ To handle the exception, we write a “try-catch” block.

₪ To pass the exception “up the chain”, we declare a throws clause in our method or class declaration.

an exception, the normal code
Wherever the code may trigger

logic is placed inside a block of

code starting with the “**try**” keyword:

After the try block, the code to handle the try

catch

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.
Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

```
try
{
//Protected code
 }
catch(ExceptionType1 e1)
{
//Catch block
 }
finally
{
//The finally block always executes.
}
```

```
try
{
// try block
}
catch (type1 arg)
{
// catch block
}
catch (type2 arg)
{
// catch block
}
catch (type3 arg)
{
// catch block
}
...
catch (typeN arg)
{
// catch block
```

throw *exception;*

Note:

› If an exception is thrown for which there is no applicable **catch** statement, an abnormal program termination may occur.

› Throwing an unhandled exception causes the standard library function

terminate() to be invoked.

› By default, **terminate()** calls **abort()** to stop the program, but we can specify our own termination handler.

In any method that might throw an exception, you may declare the method as "throws" that exception, and thus avoid handling the exception yourself

throws Exception

Java Exception Hierarchy

//Parameterless Constructor

//Constructor that accepts a message

e:

Usag

```
#include <iostream>
using namespace std;

int main()
{






cout << "Start\n";
try
{ // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not
execute";
}
```

(double i)

catch (int i)

```
{ // catch an error
cout << "Caught an exception --
value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

```
#include <iostream>
using namespace std;
void Xtest(int test)
{



cout << "Inside Xtest, test is: "
<< test << "\n";
if(test)
throw test;
}
int main()
{
```

```
cout << "Start\n";
try
{
cout << "Inside try block\n";
Xtest(0);
Xtest(1);
Xtest(2);
}
catch (int i)
{
cout <<"Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

Exception Handling Options

There are several additional features and nuances to C++ exception handling that make it easier and more convenient to use. These attributes are discussed here.

In some circumstances we want an exception handler to catch all exceptions instead of just a certain type.

This is easily accomplished by using this form of catch:

```
catch(...)
{
    // process all exceptions
}
```

```
ret-type func-name(arg-list) throw(type-list)
{
        //…
}
```

Note:

- only those data types contained in the comma-separated type-list may be thrown by the function

- <u>throwing any other type of expression will cause abnormal program termination</u>

- if we don't want a function to be able to throw any exceptions, then we use an empty list

What happens when an

unsupported exception is thrown?

Note:

hierarchy

¯ a function can be restricted only in what types of exceptions it throws back to the **try** block that called it.

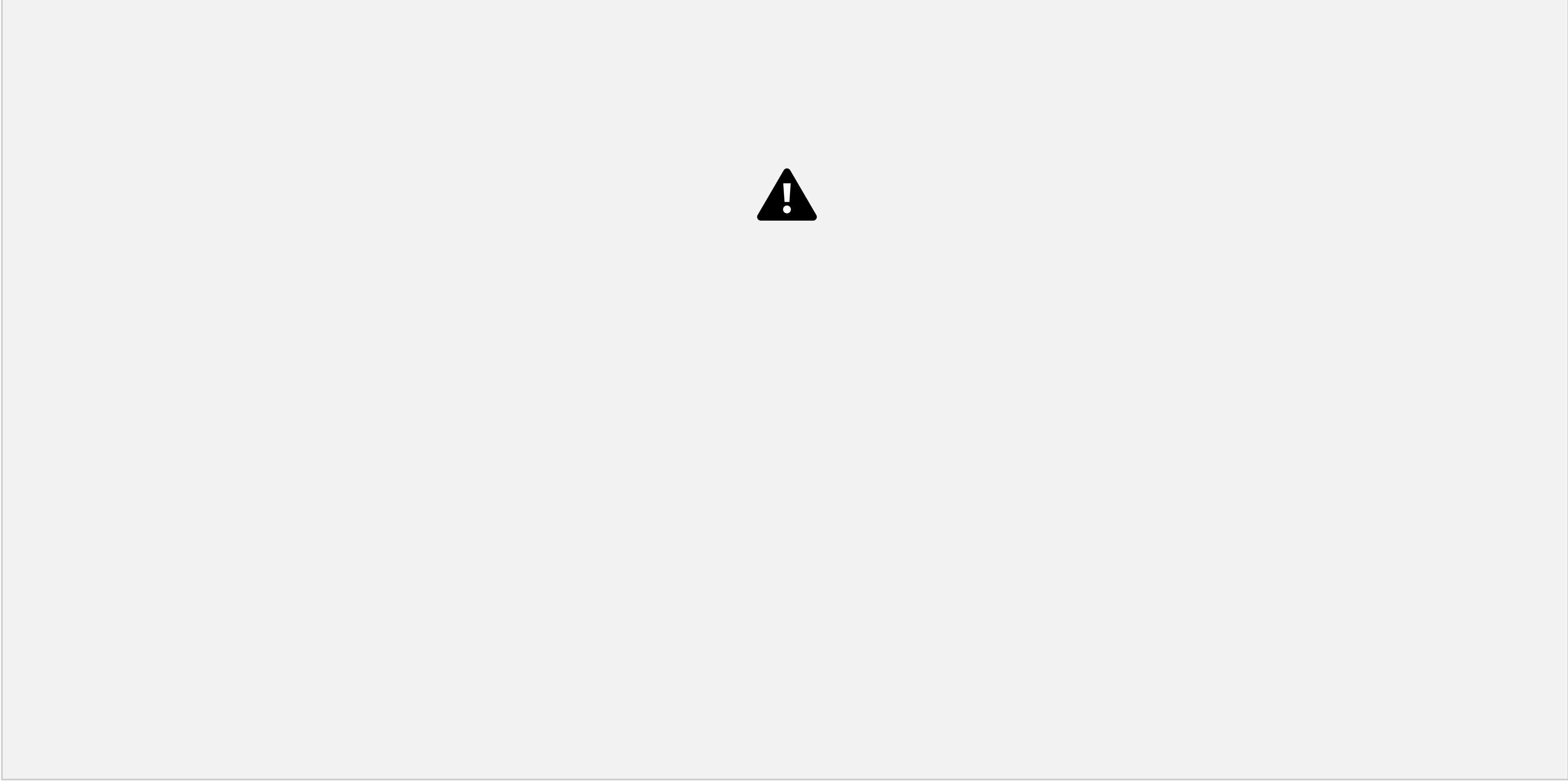
¯ a **try** block *within* a function may throw any type of exception so long as it is caught
within that function.

Note:

₪ this causes the current exception to be passed on to an outer **try/catch**
sequence
₪ when we rethrow an exception, it will be caught by the next catch statement in the

throw ;

void terminate() *void unexpected()*



terminate() is called in the following circumstances:

₪ whenever the exception handling subsystem fails to find a matching **catch** statement for an exception. ₪ It is also called if our program attempts to rethrow an exception when no exception was originally thrown. ₪ is also called under various other, more obscure circumstances.

Note:

₪ In general, **terminate()** is the handler of last resort when no other handlers for an exception are available. **By default, terminate() calls abort().**

The **unexpected()** function is called when a function attempts to throw an exception that is not allowed by its **throw** list.

By default, **unexpected()** calls **terminate()** .

To change the terminate handler, we use **set_terminate()** , shown here:

```
terminate_handler set_terminate(terminate_handler newhandler) throw(

);
```

Here, *newhandler* is a pointer to the new terminate handler. The function returns a pointer to the old terminate handler. The new terminate handler must be of type **terminate_handler**, which is defined like this:

```
typedef void (*terminate_handler) ( );
```

To change the unexpected handler, use **set_unexpected()** , shown here:

```
unexpected_handler set_unexpected(unexpected_handler newhandler) throw(

);
```

Here, *newhandler* is a pointer to the new unexpected handler. The function returns a pointer to the old unexpected handler. The new unexpected handler must be of type **unexpected_handler**, which is defined like this:

```
typedef void (*unexpected_handler) ( );
```

```
#include <cstdlib>
#include <exception>
using namespace std;

void my_Thandler()
{







cout << "Inside new
terminate handler\n";
abort();
}
int main()
{
```

```
try
{
cout << "Inside try block\n";
throw 100;
}
catch (double i)
{}
return 0;
}
```

```
set_terminate(my_Thandler);
```

```
bool uncaught_exception( );
```

Miscellaneous

Bibliography