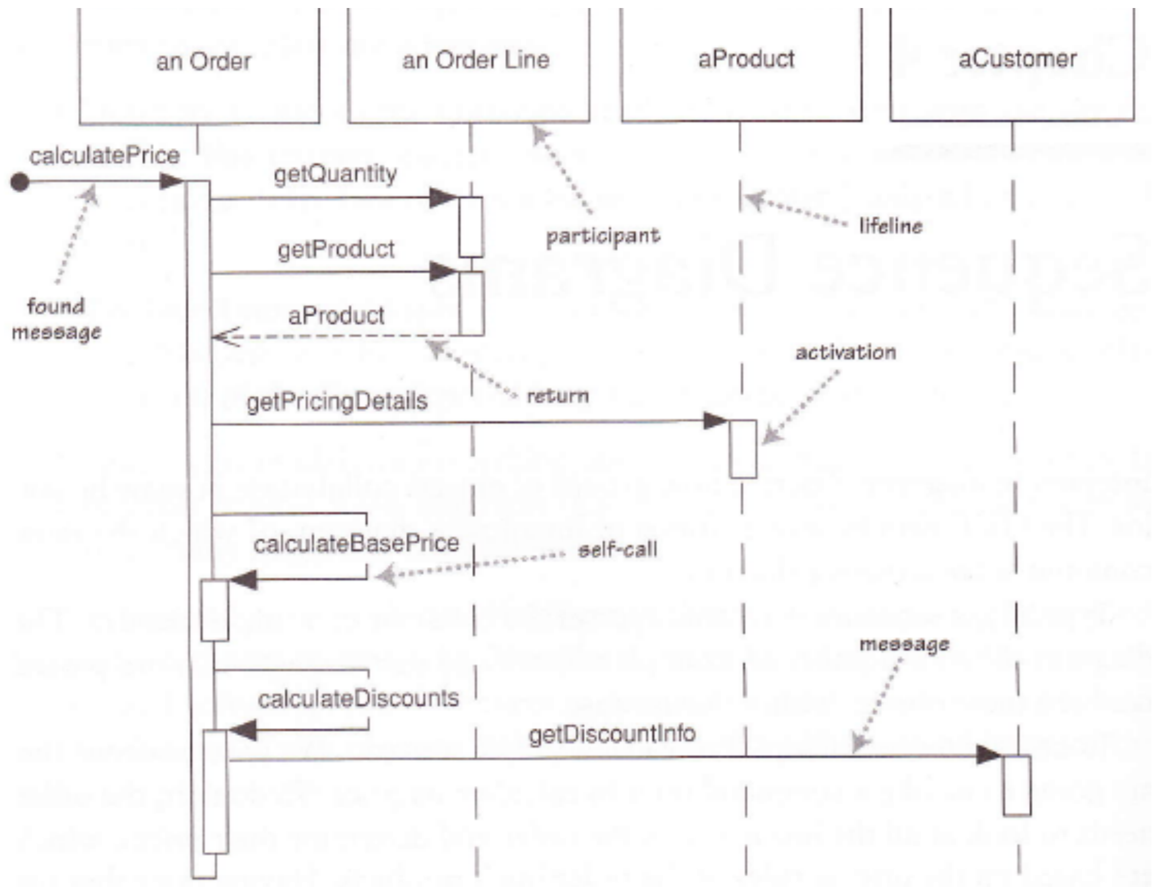
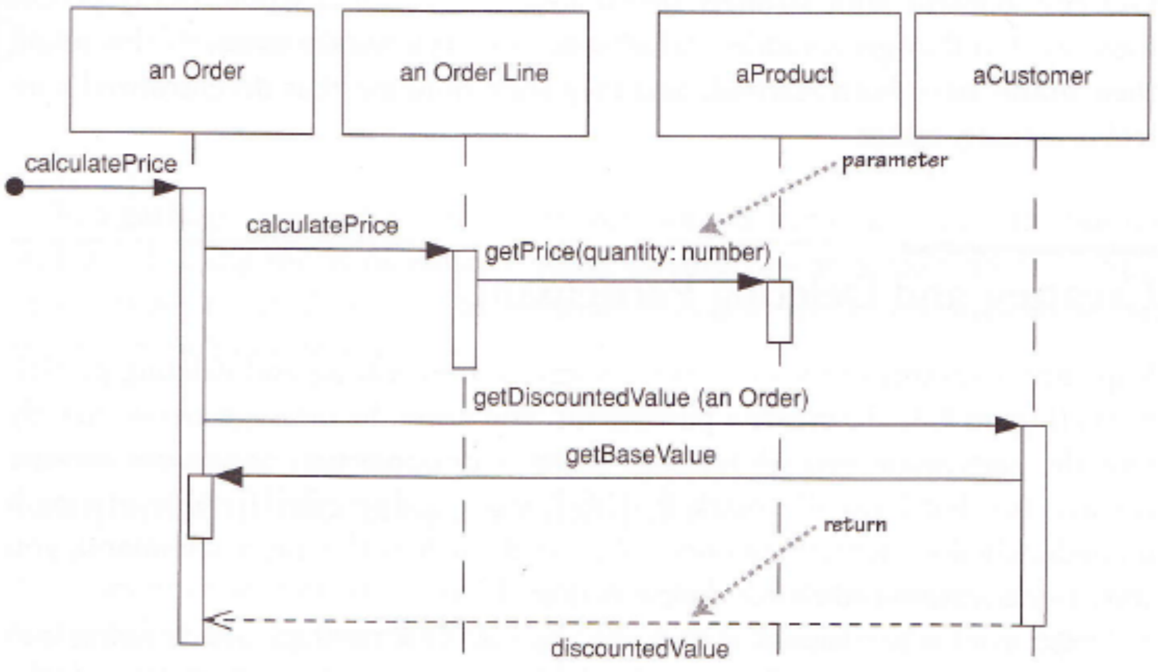
1.


an Order
an Order Line
aProduct
aCustomer
calculatePrice
getQuantity
participant
lifeline
getProduct
found message
aProduct
return
activation
getPricingDetails
calculateBasePrice
self-call
message
calculateDiscounts
getDiscountInfo

2.

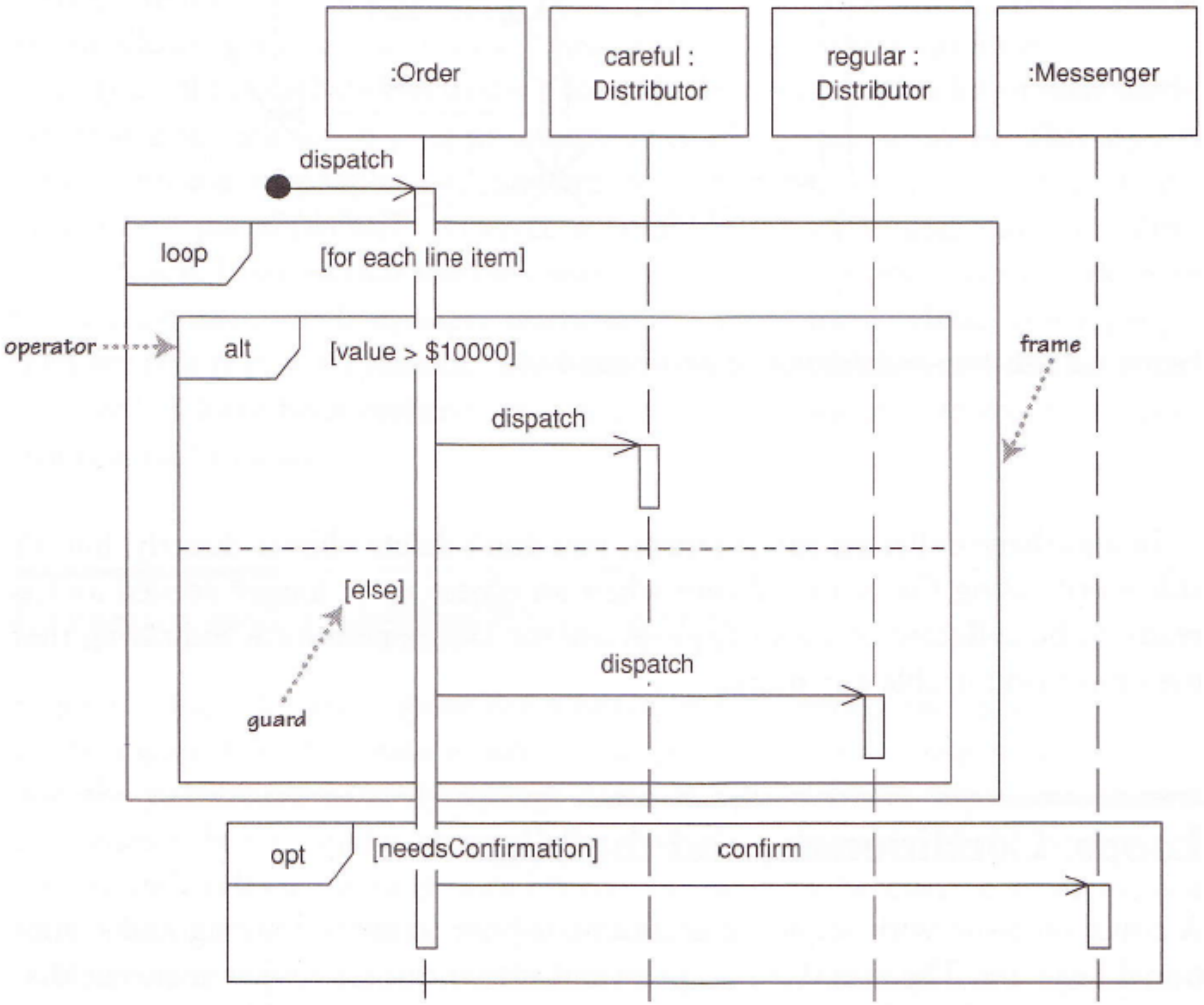

an Order
an Order Line
aProduct
aCustomer
calculatePrice
parameter
calculatePrice
getPrice(quantity: number)
getDiscountedValue (an Order)
getBaseValue
return
discountedValue

Interaction Frames

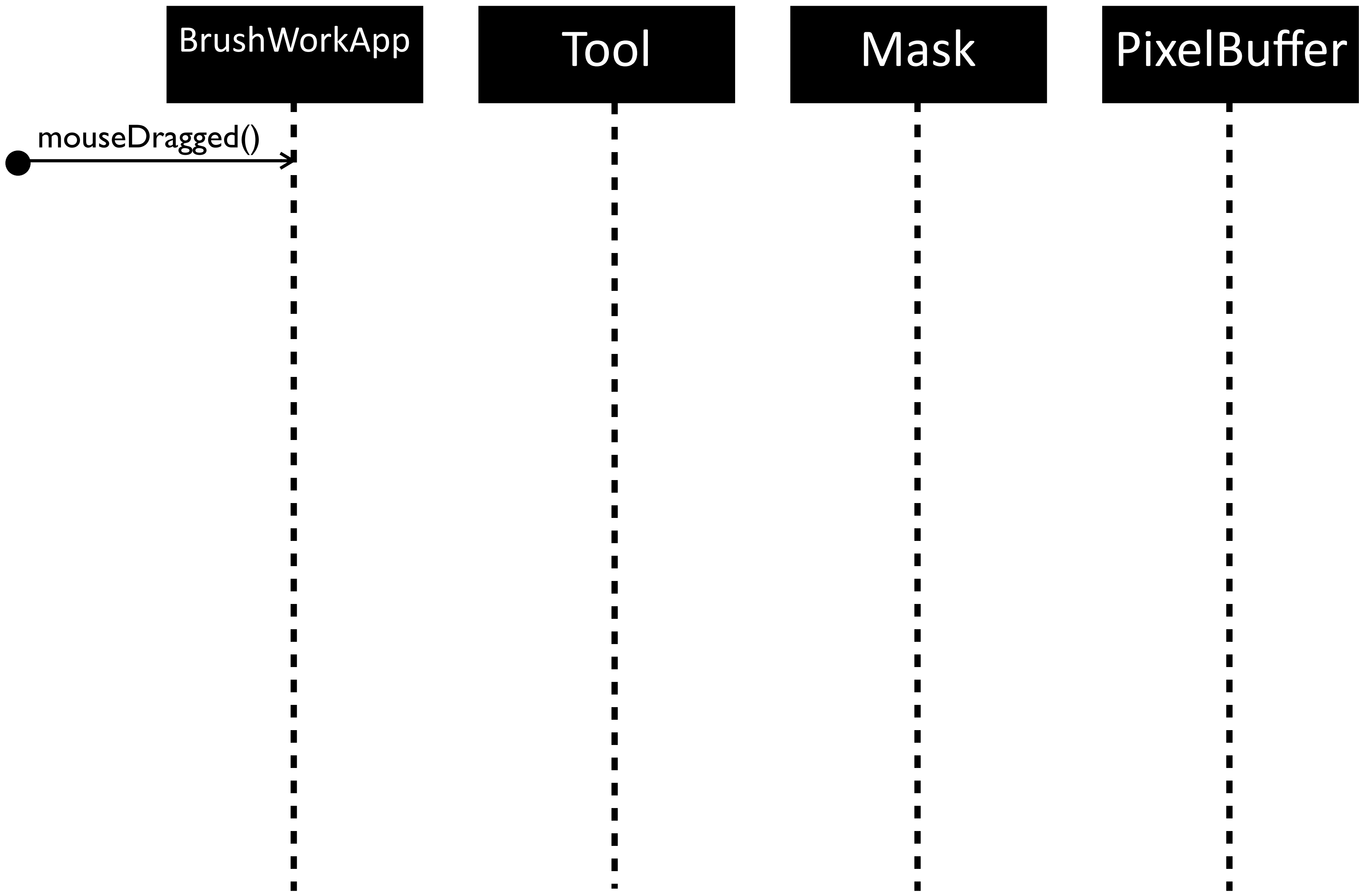
- Used for:
  - looping
  - conditionals
  - etc..

Fowler Fig. 4.4

```
procedure dispatch
  foreach (lineitem)
    if (product.value > $10K)
      careful.dispatch
    else
      regular.dispatch
    end if
  end for
  if (needsConfirmation) messenger.confirm
end procedure
```

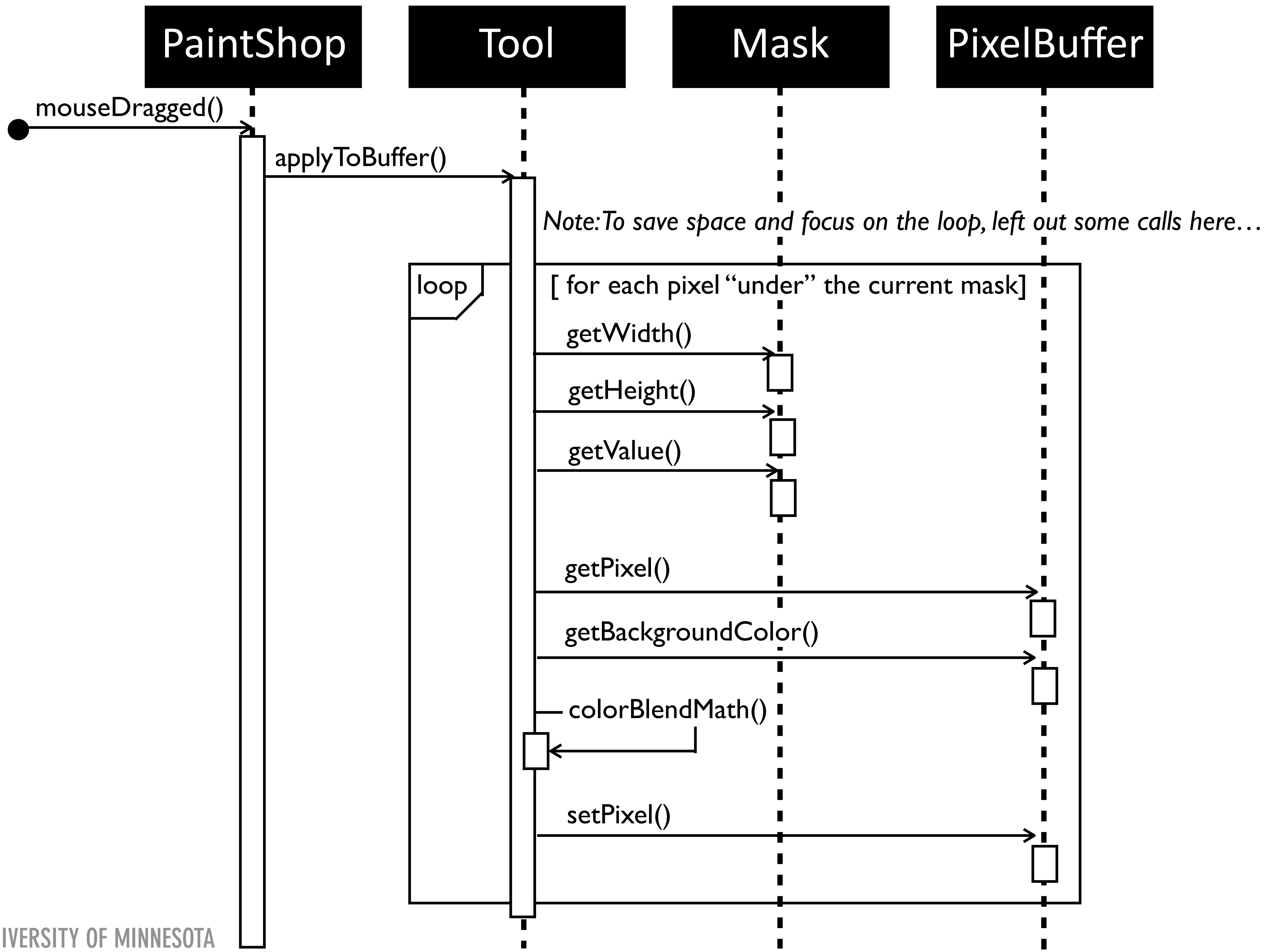


Sequence for Applying a Tool to Canvas



UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

```
void Tool::applyToBuffer(int toolX, int toolY, ColorData toolColor, PixelBuffer* buffer) {

   if (m_mask == NULL) {
      return;
   }

   int left_bound  = std::max(toolX - m_mask->getWidth()/2, 0);
   int right_bound = std::min(toolX + m_mask->getWidth()/2, buffer->getWidth()-1);
   int lower_bound = std::max(toolY - m_mask->getHeight()/2, 0);
   int upper_bound = std::min(toolY + m_mask->getHeight()/2, buffer->getHeight()-1);

   for (int y = lower_bound; y <= upper_bound; y++) {
      for (int x = left_bound; x <= right_bound; x++) {
       int mask_x = x - (toolX - m_mask->getWidth()/2);
       int mask_y = y - (toolY - m_mask->getHeight()/2);
       float mask_value = m_mask->getValue(mask_x, mask_y);

       ColorData current = buffer->getPixel(x, y);

       float slimmed_mask_value = powf(mask_value,3);
       ColorData c = colorBlendMath(slimmed_mask_value, toolColor, current,
                                    buffer->getBackgroundColor());

       buffer->setPixel(x, y, c);
      }
   }

}
```



UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Putting these ideas together…

Using diagrams effectively in your Iteration #3 writing

- What types of diagrams can you use and where?
- What guidelines can you infer from our uses of diagrams and figures in class?
- How about from the examples you found online today at the beginning of class?
- How do you weight the importance of these diagrams relative to writing in a more narrative form?

Defensive Programming:
Assertions, Exceptions, Handling Errors in C++

UNIVERSITY OF MINNESOTA

PROFESSOR DANIEL F. KEEFE

How to be a defensive programmer

- When somebody says your program doesn’t work, what do you reply?
- “It does so work!”

Defensive Programming

- A worthy goal:
- Protect yourself from the cold, cruel world of invalid data, events that can “never” happen, and other programmers’ mistakes.

Topics

- Invalid Inputs
- Assertions
- Error-Handling Techniques
- Debugging Aids

Invalid Inputs

- Check the values of all data from external sources
  - Sources: a file, a user, the network, other external interface
  - Types of data: numeric values, strings, input buffers, system calls, etc.
- Check the values of all routine input parameters
  - Same as above, except the data come from another routine instead of an external interface.
- Decide how to handle bad inputs
  - Lots of choices...

Assertions

- Code that is used during development--usually a routine or a macro--that allows a program to check itself as it runs.
- When an assertion is true, that means everything is operating as expected.
- When it’s false, that means it has detected an unexpected error in the code.

Assertions (2)

- In C++ the built-in assert(); just takes one argument
  - a boolean expression that describes the assumption that's supposed to be true.
- But, many assertion macros take two arguments
  - a boolean expression that describes the assumption that's supposed to be true.
  - a message to display if it isn't

```
void assert (int expression);
```

If the expression is false, a message is written to the standard error device and abort is called, terminating the program execution.

```
#include <stdio.h>      // printf
#include <assert.h>     // assert

void print_number(int* myInt) {
  assert(myInt != NULL);
  printf("%d\n",*myInt);
}
```

The specifics of the message at least include: the expression whose assertion failed, the name of the source file, and the line number where it happened:

```
Assertion failed: expression, file filename, line line number
```

The macro is automatically disabled when NDEBUG is defined, so when you compile your code in "release mode", you should define this flag so all the debugging assertions will not slow down your program, e.g.:

```
g++ -c -D NDEBUG myprogram.cpp
```

Possible Things to Assert

- That an input (or output) parameter's value falls within its expected range.
- That a file or stream is open when a routine begins executing.
- That a file or stream is at the beginning when a routine starts.
- That the value of an input-only variable is not changed by a routine.
- That a pointer is non-NULL.
- That an array or other container passed into a routine can contain at least X number of data elements.
- That a table has been initialized to contain real values.
- That a container is empty when a routine starts.
- That the results from a highly optimized, complicated routine match the results from a slower but clearly written routine.

Boy, this sure sounds a lot like testing… what’s the difference?

Guidelines

- Use error-handling for conditions you expect to occur; use assertions for conditions that should never occur. Why?
- Avoid putting executable code into assertions.
  - this is bad:

```
assert( PerformAction() );
```

  - this is good:

```
bool actionPerformed = PerformAction();
assert( actionPerformed );
```

Guidelines (2)

- Use assertions to document and verify preconditions and postconditions.
- Preconditions: properties that the client code promises will be true before it calls the routine.
- Postconditions: properties that the routine promises will be true when it concludes executing.

Example of Pre- and Post-Conditions

```
float velocity(float latitude, float longitude, float elevation) {
  // Preconditions
  assert((-90 <= latitude) && (latitude <= 90));
  assert((0 <= longitude) && (longitude < 360));
  assert((-500 <= elevation) && (elevation < 75000));

  // Do some computation...
  ...

  // Postconditions
  assert((0 <= returnVel) && (returnVel <= 600));
  return returnVel;
}
```

Guidelines (3)

- For highly robust code, assert and then handle the error anyway...
- The error handling code will only be reached in the release version, the debug version will quit the program when the assertion fails.
- (This strategy is used inside Microsoft Word.)

```
float velocity(float latitude, float longitude, float elevation) {
  // Preconditions
  assert((-90 <= latitude) && (latitude <= 90));
  assert((0 <= longitude) && (longitude < 360));
  assert((-500 <= elevation) && (elevation < 75000));

  // Sanitize the input data.  Values should be within the ranges
  // asserted above, but if a value is not, it will be changed to
  // the closest legal value.
  if (latitude < -90)
    latitude = -90;
  else if (latitude > 90)
    latitude = 90;
  if (longitude < 0)
    longitude = 0;
  else if (longitude >= 360)
    longitude = 0;
  ...
```

Customizing Assert

Here’s a nice example:
http://www.go4expert.com/forums/showthread.php?t=18309

```
#define DEBUG

#include<iostream.h>

#ifndef DEBUG
  // Do nothing when DEBUG is not defined
  #define MY_ASSERT(X) void(0)
#else
  #define MY_ASSERT(x) \
    logMsg(x, #x, __FILE__, __LINE__)
#endif


bool logMsg(bool x, const char *msg, char* file, unsigned int line) {
  if (false == x) {
    cout << "On line " << line << “:" ;
    cout << " in file " << file << ":";
    cout << " Error !! Assert “ << msg << " failed\n";
    abort();
    return (true);
  }
  else {
    return (true);
  }
}
```

Defensive Programming Key Points

- The fallback plan, “don’t blame the programmers, if you put garbage in, you get garbage out”, is not appropriate for production code. So, we need to use defensive programming techniques.
- Defensive programming makes errors easier to find, easier to fix, and less damaging to production code.
- Assert is a great method to help us detect programming mistakes.
- What about bad inputs?
- One of your important high-level design decisions will need to be how to handle bad inputs… there are many ways to do it, so this is an important ***design*** decision to make.

Possible Error Handling Techniques

- Return a neutral value
- Substitute the next piece of valid data
- Return the same answer as the previous time
- Substitute the closest legal value
- Log a warning message to a file
- Return an error code
- Call an error-processing routine/object
- Display an error message whenever the error is encountered
- Handle the error in whatever way works best locally
- Shut down

Intro to Exceptions

- Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.
- To catch exceptions, a portion of code is placed under exception inspection.
- When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

http://www.cplusplus.com/doc/tutorial/exceptions/

Simple Exception Example

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

int main () {
  try
  {
    // some complex code
    throw 20; // “throw” an exception if the complex code failed
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << endl;
  }
  return 0;
}
```

http://www.cplusplus.com/doc/tutorial/exceptions/

You can “throw” anything: int, string, etc..

```
try {
  throw string("Example Error");
}
catch (string s)
{
  cout << "Error occurred: " << s << endl;
}
```

- Exceptions are a very powerful error handling mechanism.
- They can also drive you crazy… be very careful with the order of execution for exceptions…

“catch” only catches exceptions of the correct type.

```
try {
  line 1 of C++ code
  line 2
  if (line_2_failed) throw 10;
  line 4
  if (line_4_failed) throw ‘f’;
  line 6
  line 7
  if (line_7_failed) throw string(“line 7 failed”);
  line 9
  line 10
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

http://www.cplusplus.com/doc/tutorial/exceptions/

You can have nested try-catch blocks

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;    ← Forwards the exception on to the outer block
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

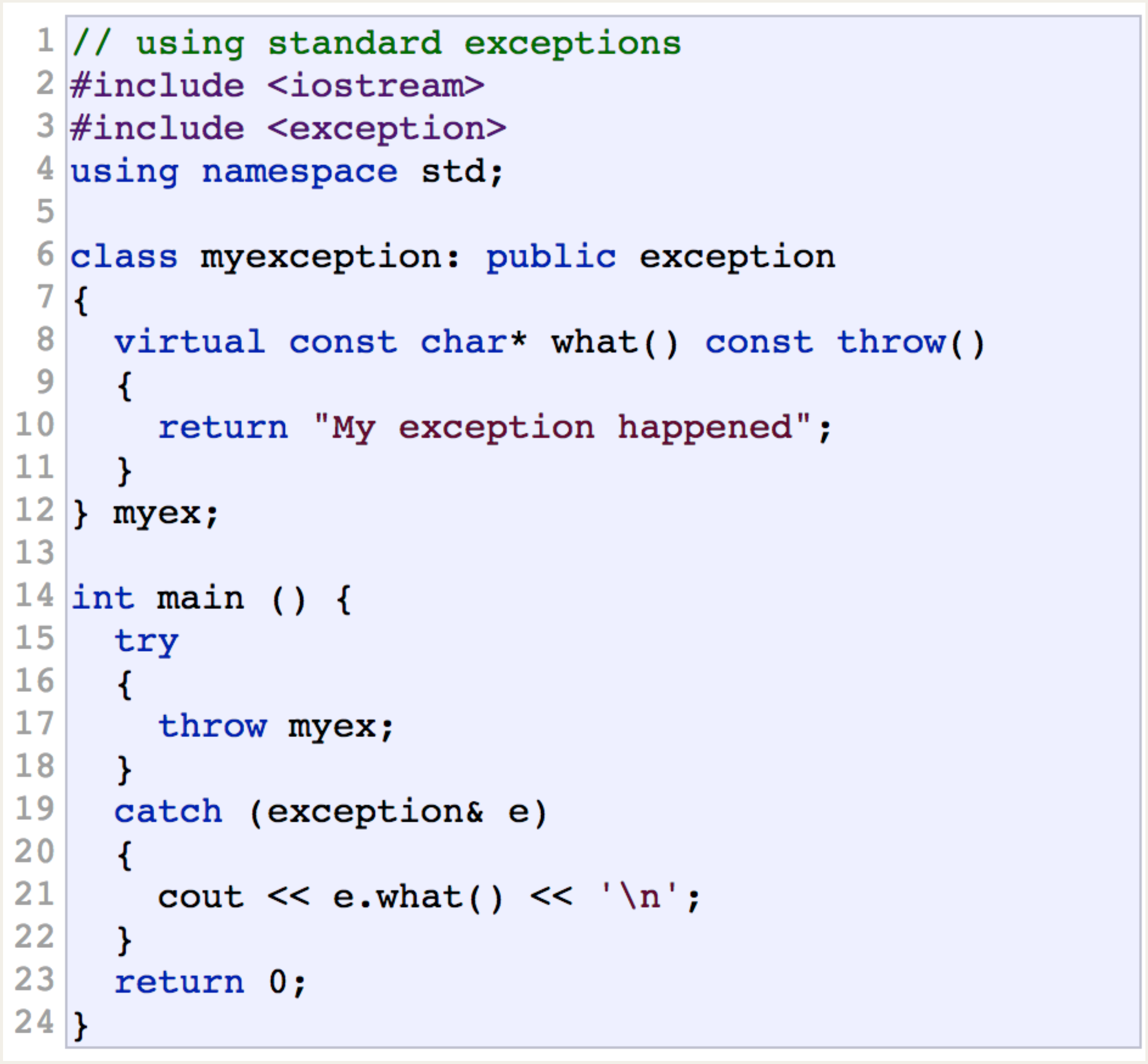
- How could you use this?
- Note how powerful this is… but also possibly confusing. The code no longer runs linearly.

http://www.cplusplus.com/doc/tutorial/exceptions/

The C++ standard library (std)

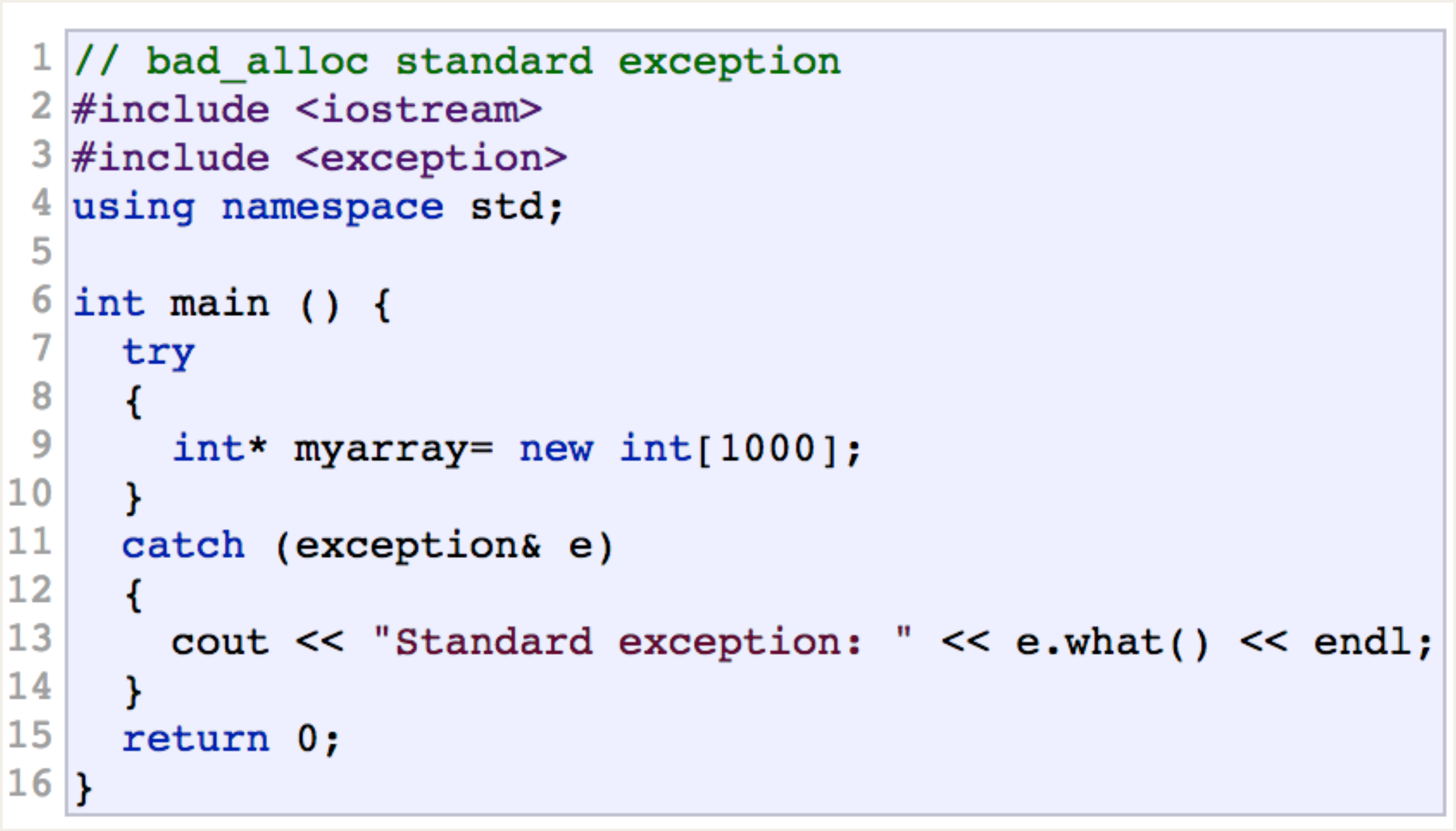
- We’ve used std for several things already:
  - string
  - cout, cerr
- std also includes a useful base class for defining exceptions.
- std uses this class itself for all of its internal error handling.
- you can also extend it and use it yourself.

Here's how to extend std::exception to create your own "myexception" class.



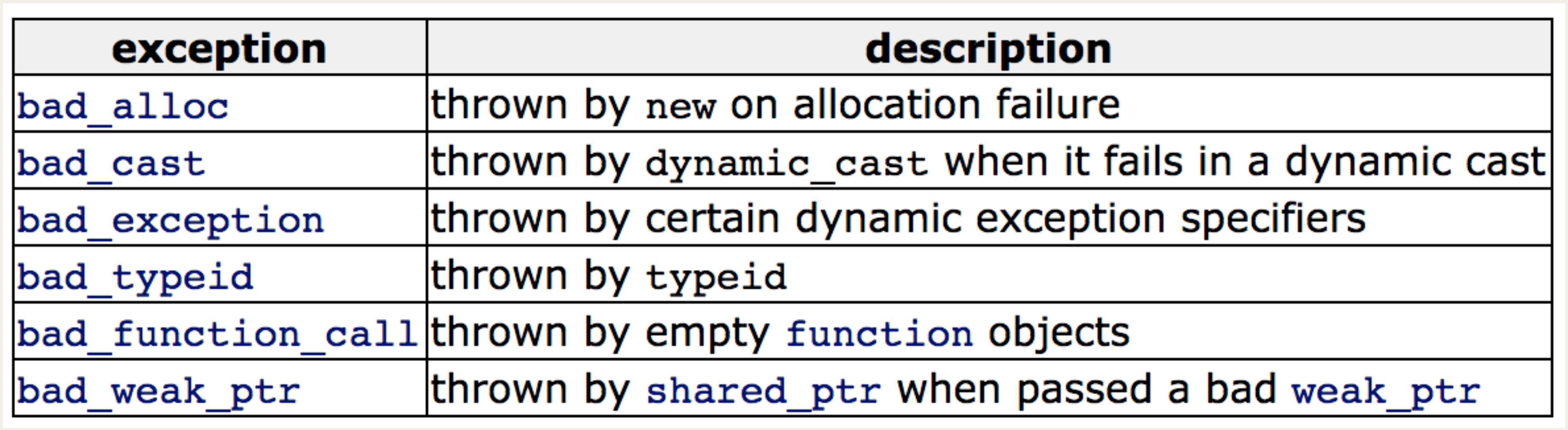
```
// using standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n';
  }
  return 0;
}
```

Here's how to catch std::exception's thrown within the std library:



```cpp
// bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;

int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " << e.what() << endl;
  }
  return 0;
}
```

Here are the different exceptions that std can throw:



exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when it fails in a dynamic cast
bad_exception	thrown by certain dynamic exception specifiers
bad_typeid	thrown by typeid
bad_function_call	thrown by empty function objects
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr

Remember, exceptions are just a programmatic way to catch errors, you still need to handle them in some way.

- Return a neutral value
- Substitute the next piece of valid data
- Return the same answer as the previous time
- Substitute the closest legal value
- Log a warning message to a file
- Return an error code
- Call an error-processing routine/object
- Display an error message whenever the error is encountered
- Handle the error in whatever way works best locally
- Shut down