Engineering Robust Server Software

Exceptions

Duke UNIVERSITY

Andrew Hilton / Duke ECE

Exceptions

- Handling problems: exceptions
- C++
  - temp-and-swap
  - RAII
  - Smart Pointers
- Java
  - finally
  - specifications
  - finalizers (and why they are not what you need for this)

Exceptions

- Review: exceptions = way to handle problems
  - Thing goes wrong? throw exception
  - Know how to deal with problem? try/catch exception
    - In python, try/except
- Why exceptions?
  - Return error code? Cluttery, easy to forget/ignore
  - Do nothing? Automatically pass problem to caller
  - Provide details about error

Exceptions: Downsides

- So exceptions: best idea ever?
- Downsides too
  - Unexpected things happen in code
    - Well, that is true anyways
  - Used improperly: corrupted objects, resource leaks, …
- Bottom line:
  - Good if you do all things right

Exception Safety

- Continued review: exception safety
  - Remind us of the four levels of exceptions safety?

Stronger Guarantees

No Throw	Will not throw any exception. Catches and handles any exceptions throw by operations it uses
Strong	No side-effects if an exception is thrown: objects are unmodified, and no memory is leaked
Basic	Objects remain in valid states: no dangling pointers, invariants remain intact. No memory is leaked
None	Does not provide even a basic exception guarantee. Unacceptable in professional code.

Exception Safety

```
template<typename T>
class LList {
  //other things omitted, but typical

  LList & operator=(const LList & rhs) {
     if (this != &rhs) {
        deleteAll();
        Node * curr = rhs.head;
        while (curr != null) {
           addToBack(curr->data);
           curr = curr->next;
        }
     }
     return *this;
  }
};
```

Which guarantee does this make?

Exception Safety

```
template<typename T>
class LList {
  //other things omitted, but typical

  LList & operator=(const LList & rhs) {
    if (this != &rhs) {
      deleteAll();
      Node * curr = rhs.head;
      while (curr != null) {
        addToBack(curr->data);
        curr = curr->next;
      }
    }
    return *this;
  }
};
```

Which guarantee does this make?
- Need to know what guarantees these make!

Exception Safety

```
template<typename T>
class LList {
  //other things omitted, but typical

  void deleteAll(){
    while(head != nullptr) {
       Node * temp = head->next;
       delete head;
       head = temp;
    }
    tail = nullptr;
  }
};
```

Which guarantee does deleteAll() make?

No throw guarantee
(assuming destructors are no-throw)

Exception Safety

```
template<typename T>
class LList {
  //other things omitted, but typical

  void addToBack(const T& d){
    Node * newNode = new Node(d, nullptr, tail);
    if (tail = nullptr) {
      head = tail = newNode;
    }
    else {
      tail->next = newNode;
      newNode->prev = tail;
      tail = newNode;
    }
  }
};
```

Which guarantee does addToBack() make?

Depends on copy constructor for T

T's Copy Constructor	addToBack()
No Throw	Strong
Strong	Strong
Basic (??)	Basic
No Guarantee	No Guarantee

Exception Safety

Which guarantee does this make?

Basic

```
template<typename T>
class LList {
  //other things omitted, but typical

  LList & operator=(const LList & rhs) {
    if (this != &rhs) {                 //no throw
      deleteAll();                      //no throw
      Node * curr = rhs.head;           //no throw
      while (curr != null) {            //no throw
        addToBack(curr->data);          //strong [let us suppose]
        curr = curr->next;              //no throw
      }
    }
    return *this;                       //no throw
  }
};
```

```
LList & operator=(const LList & rhs) {
    if (this != &rhs) {
        Node * temp = rhs.head;
        Node * n1 = nullptr;
        Node * n2 = nullptr;
        if (temp != nullptr){
            n1 = n2 = new Node(temp->data, nullptr,nullptr);
            temp = temp->next;
            while (temp != null) {
                n2->next = new Node(temp->data, n2, nullptr);
                n2 = n2->next;
                temp = temp->next;
            }
        }
        deleteAll();
        head = n1; tail = n2;
    }
    return *this;
}
};
```

Which guarantee does this version make?

No guarantee! :(

Exception Safety

Which guarantee does this make?

```
template<typename T>
class LList {
  //other things omitted, but typical

  LList & operator=(const LList & rhs) {
     if (this != &rhs) {
        LList temp(rhs);
        std::swap(temp.head, head);
        std::swap(temp.tail, tail);
     }
     return *this;
  }
};
```

Strong!

Exception Safety

```
template<typename T>
class LList {
  //other things omitted, but typical

  LList & operator=(LList rhs) {
     std::swap(temp.head, head);
     std::swap(temp.tail, tail);
     return *this;
  }
};
```

Same principle, but passed by value instead of reference
(going to copy anyways…Will waste work if self assignment)

Temp-and-swap

- Common idiom for strong guarantees: temp-and-swap
    - Make `temp` object
    - Modify `temp` object to be what you want `this` to be
    - swap fields of `temp` and `this`
    - `temp` destroyed when you return (destructor cleans up state)
        - Exception? `temp` destroyed in stack unwinding
- Downside?
    - Change only some state: may be expensive to copy entire object

What About This Code…

```
template<typename T>
class LList {
  //other things omitted, but typical

  LList & operator=(const LList & rhs) {

     if (this != &rhs) {
       m.lock();
       rhs.m.lock();        //What if this throws?
       std::swap(temp.head, head);
       std::swap(temp.tail, tail);
       rhs.m.unlock();
       m.unlock();
     }
     return *this;
  }
};
```

How About Now?

```
template<typename T>
class LList {
  //other things omitted, but typical

  LList & operator=(const LList & rhs) {

     if (this != &rhs) {
       std::lock_guard<std::mutex> lck1(m);      //calls m.lock()
       std::lock_guard<std::mutex> lck2(rhs.m); //calls rhs.m.lock()
       std::swap(temp.head, head);
       std::swap(temp.tail, tail);
     }
     return *this;
  }
};
```

How About Now?

```
template<typename T>
class LList {
  //other things omitted, but typical

  LList & operator=(const LList & rhs) {

     if (this != &rhs) {
       std::lock_guard<std::mutex> lck1(m);      //calls m.lock()
       std::lock_guard<std::mutex> lck2(rhs.m); //calls rhs.m.lock()
       std::swap(temp.head, head);
       std::swap(temp.tail, tail);
     } //destruction calls .unlock()
     return *this;
  }
};
```

Duke UNIVERSITY

How About Now?

```
template<typename T>
class LList {
  //other things omitted, but typical

  LList & operator=(const LList & rhs) {

     if (this != &rhs) {
       std::lock_guard<std::mutex> lck1(m);
       std::lock_guard<std::mutex> lck2(rhs.m);  //what if exn?
       std::swap(temp.head, head);
       std::swap(temp.tail, tail);
     }
     return *this;
  }
};
```

RAII

- Resource Acquisition Is Initialization
  - Resource lifetime tied to object lifetime
  - Allocation during initialization
  - Released during destruction
- Example resources:
  - Mutex: lock/unlock
  - Heap Memory: new/delete
  - File: open/close
- Exception safety benefits?

RAII with Heap Objects

- "Smart Pointers"
  - Objects that wrap pointer and provide RAII
  - C++03: std::auto_ptr (deprecated)
- C++11:
  - std::unique_ptr
  - std::shared_ptr
  - std::weak_ptr

std::unique_ptr

```
{
 std::unique_ptr<Thing> thing1 (new Thing);
  //other code here



} //thing1 goes out of scope: delete its pointer
```

- Owns a pointer
  - When destroyed, deletes owned pointer

std::unique_ptr

```
{
 std::unique_ptr<Thing> thing1 (new Thing);
  //other code here
  Thing * tp = thing1.get();


}
```

- Owns a pointer
  - When destroyed, deletes owned pointer
- Can use .get() to get raw pointer

std::unique_ptr

```
{
 std::unique_ptr<Thing> thing1 (new Thing);
  //other code here
  Thing * tp = thing1.get();
  thing1->doSomething();

}
```

- Owns a pointer
    - When destroyed, deletes owned pointer
- Can use .get() to get raw pointer
- Can also use * and -> operators

std::unique_ptr

```
{
 std::unique_ptr<Thing> thing1 (new Thing);
 //… … …
 std::unique_ptr<Thing> thing2 (thing1);
       //thing2 owns pointer, thing1 is empty (holds nullptr)
}
```

- Assignment operator/copy constructor **transfer** ownership

Exception Safety

Which guarantee does this make?

```
Thing * foo(int x, char c) {
   Widget * w = new Widget(x);
   Gadget * g = new Gadget(c);
   Thing * t = new Thing(w,g);
   return t;
}
```

Exception Safety

```
Thing * foo(int x, char c) {
    std::unique_ptr<Widget> w (new Widget(x));
    std::unique_ptr<Gadget> g (new Gadget(c));
    Thing * t = new Thing(w.get(),g.get());
    return t;
}
```

Is this code correct?

Exception Safety

```
Thing * foo(int x, char c) {
    std::unique_ptr<Widget> w (new Widget(x));
    std::unique_ptr<Gadget> g (new Gadget(c));
    Thing * t = new Thing(w.get(),g.get());
    return t;
}
```

w and g go out of scope here, so… what happens to their pointers?

Is this code correct? No!

Exception Safety

```
Thing * foo(int x, char c) {
    std::unique_ptr<Widget> w (new Widget(x));
    std::unique_ptr<Gadget> g (new Gadget(c));
    Thing * t = new Thing(w.release(),g.release());
    return t;
}
```

What about this code?

release returns the pointer (like get),
but also gives up ownership (sets the owned pointer to nullptr)

Exception Safety

```
Thing * foo(int x, char c) {
    std::unique_ptr<Widget> w (new Widget(x));
    std::unique_ptr<Gadget> g (new Gadget(c));
    Thing * t = new Thing(w.release(),g.release());
    return t;
}
```

What if **new** fails?

What about this code?

"Whether the allocation function is called before evaluating the constructor arguments or after evaluating the constructor arguments but before entering the constructor is unspecified. It is also unspecified whether the arguments to a constructor are evaluated if the allocation function returns the null pointer or exits using an exception. "

— C++ standard, 5.3.4 (21)

Exception Safety

```
Thing * foo(int x, char c) {
    std::unique_ptr<Widget> w (new Widget(x));
    std::unique_ptr<Gadget> g (new Gadget(c));
    Thing * t = new Thing(w, g);
    return t;
}
```

What about this code?
(What am I assuming Thing's constructor takes now?)

Shared Pointers + Weak Pointers

- Unique Pointers: exactly one owner
  - Assignment **transfers** ownership
- Shared Pointers: many owners
  - Copying increments count of owners
  - Destruction decrements counts of owners
  - Object freed when owner count reaches 0
- Weak Pointers: non-owners of shared pointer
  - Can reference object, but does not figure into owner count
  - Use .lock() to obtain shared_ptr: has object (if exists) or nullptr (if not)

Java Exceptions: Slightly Different

- RAII: C++, but not Java (why not?)
  - No objects in stack in Java (all in heap…)
- Java's plan: finally
  - **ALWAYS** executed, no matter whether exception or not

Java Exceptions: Slightly Different

```
public void doAThing(String name) {
   SomeResource sr = null;
   try {
      sr = new SomeResource(name);
      doStuff(sr);
   }
   catch(WhateverException we) {
      dealWithProblem(we);
   }
   finally {
      if(sr != null) {
          sr.close();
      }
   }
}
```

Java Exceptions: Slightly Different

```
public void doAThing(String name) throws WhateverException{
   SomeResource sr = null;
   try {
      sr = new SomeResource(name);
      doStuff(sr);
   }
   finally {
      if(sr != null) {
          sr.close();
      }
   }
}
```

Can have try-finally (no catch)
- Allows exception to propagate out
- Cleans up resources

Java Exceptions: Slightly Different

```
public void doAThing(String name) throws WhateverException{
  try (SomeResource sr = new SomeResource(name)) {
      doStuff(sr);
  }
 }
```

Java also has try-with-resource
- declare/initialize AutoCloseable object in () after try
  - o can have multiple declarations, separate with ;
- automatically makes a finally which closes it
  - o closes in reverse order of creation
- can have explicit catch or finally if you want

Java Exceptions: Slightly Different

```
public void doAThing(String name) throws WhateverException{
   SomeResource sr = null;
   try {
      sr = new SomeResource(name);
      doStuff(sr);
   }
   finally {
      if(sr != null) {
          sr.close();
      }
   }
}
```

Java's exception specification rules different from C++'s

Exception Specifications

- C++ 03
    - No declaration: can throw anything
    - Declaration: restricted to those types   throw(x, y, z)   or throw()
        - Checked at runtime: when exception is thrown
        - If lied, std::unexpected()

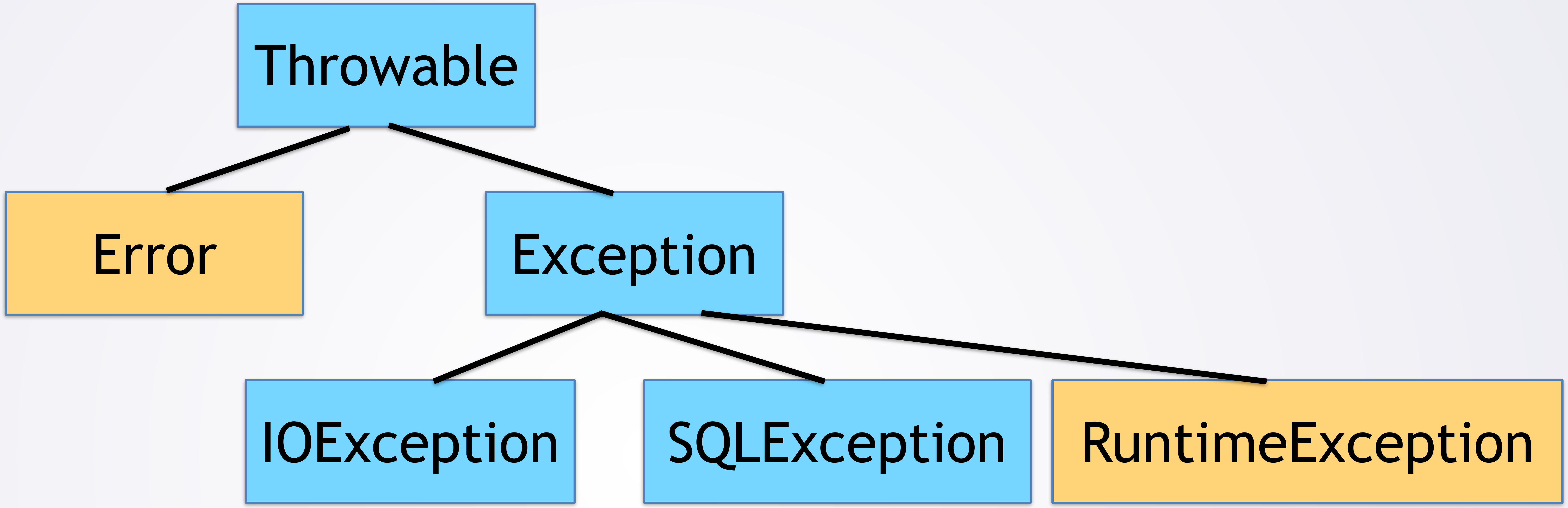
Exception Specifications

```
template<typename T>
class Thing {
   T data;
   public:
     Thing() noexcept(noexcept(T()))  {}
//............
};
```

- C++ 11
  - C++03 specifications valid but deprecated
  - noexcept for "no throw"
    - Can take a **boolean expression** to indicate behavior (true=noexcept)
    - **noexcept(expr)** queries if expr is declared noexcept
  - If noexcept actually throws, calls std::terminate()

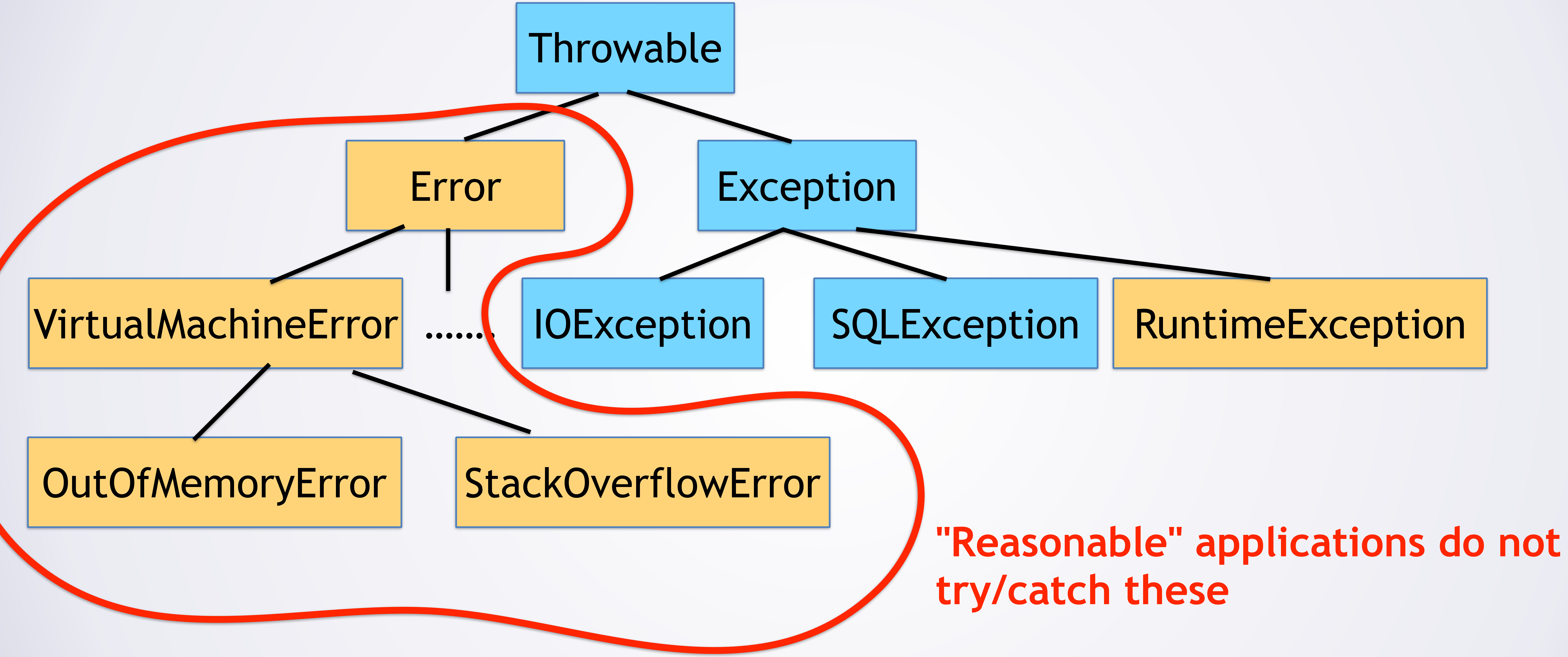
Duke UNIVERSITY

Exception Specifications



- Java
  - Two types of exceptions: checked and unchecked
  - **Checked**: exception specifications checked at compile time
    - Compiler ensures you don't lie (aka miss one)
  - **Unchecked**: no need to declare in spec
    - Possible in too many places, would clutter code

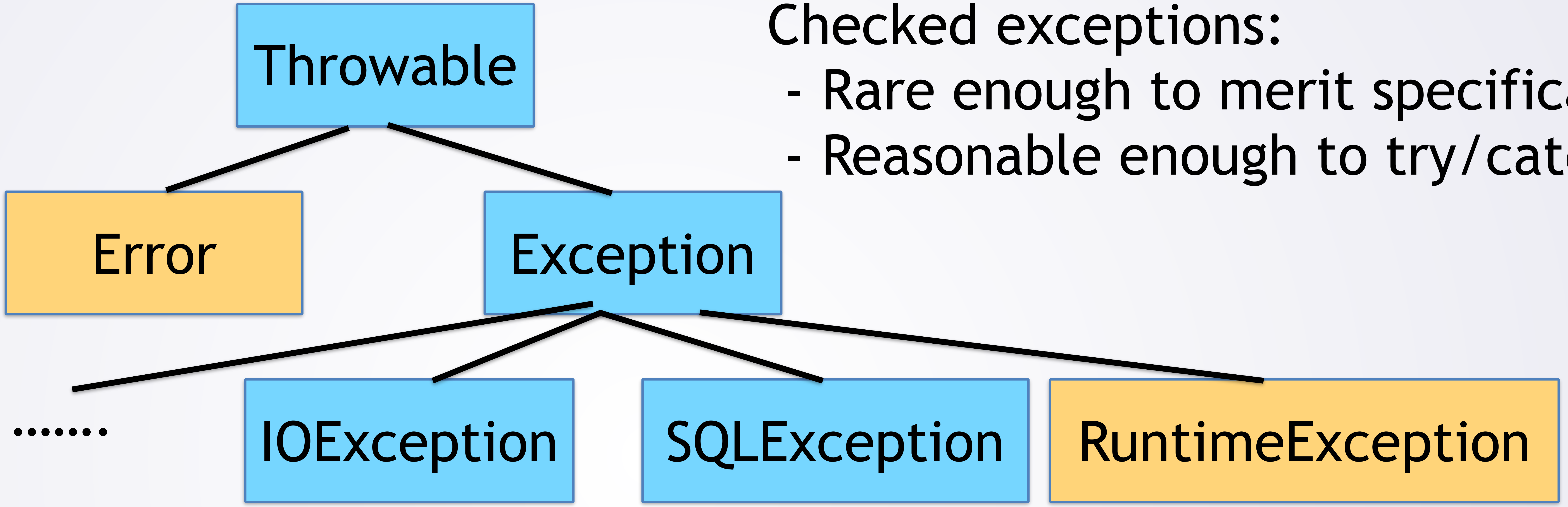
Exception Specifications



Exception Specifications

RuntimeException:
too ubiquitous to clutter
code with specifications
(everything might throw them)

Exception Specifications



Checked exceptions:

- Rare enough to merit specification
- Reasonable enough to try/catch

Java: Finalizers

- Java objects have .finalize()
    - "Called by the garbage collector on an object when garbage collection determines that there are no more references to the object."
- Seems like maybe we could use this to help resource management?

Lets Look at Stack Overflow

When the IO resource
method.

JVM) before the program exits
Multi-threaded programs
do not face resource
od closes the Socket

http://stackoverflow.com/questions/12958440/closing-class-io-resources-in-overridden-finalize-method
http://stackoverflow.com/questions/8051863/how-can-i-close-the-socket-in-a-proper-way

Finalizer: NOT For Resource Management

- Do NOT try to use finalizers for resource management!
  - No guarantee of when they will run (may never gc object!)
- Do NOT use finalizers in general
  - May run on other threads (possibly multiple finalizers at once)
    - Were you thinking about how to synchronize them?
    - What about deadlock?
  - Likely to run when memory is scarce (may cause problems if you allocate)
  - Could accidentally make object re-referenceable?

Exceptions

- Handling problems: exceptions
- C++
  - temp-and-swap
  - RAII
  - Smart Pointers
- Java
  - finally
  - specifications
  - finalizers (and why they are not what you need for this)