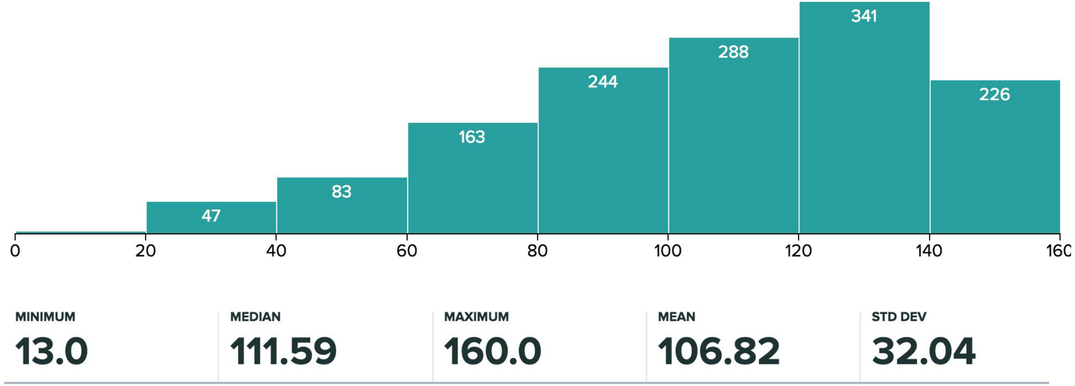
Announcements

Midterm grades out:



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
13.0	111.59	160.0	106.82	32.04

See @1520.

If you were significantly below median and hope for a B+, please contact the staff (@1497) to discuss whether or not you should drop the course.

- No turning back after today!

Regrades will open this Sunday, due by next Friday.

Announcements

Project 2:

- It will be hard. You should take it seriously.
- Highly recommended to work with a partner

CS61B: 2018

Lecture 14: Exceptions, Iteration

- Exceptions
- Iterations

datastructur.es

Exceptions

Exceptions

Basic idea:

- When something goes really wrong, break the normal flow of control.
- So far, we’ve only seen implicit exceptions, like the one below.

```java
public static void main(String[] args) {
   ArrayMap<String, Integer> am = new ArrayMap<String, Integer>();
   am.put("hello", 5);
   System.out.println(am.get("yolp"));
}
```

```
$ java ExceptionDemo
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: -1
    at ArrayMap.get(ArrayMap.java:38)
    at ExceptionDemo.main(ExceptionDemo.java:6)
```

Explicit Exceptions

We can also throw our own exceptions using the **`throw`** keyword.

- Can provide more informative message to a user.
- Can provide more information to some sort of error handling code.

```
public V get(K key) {
   int location = findKey(key);
   if (location < 0) { throw new IllegalArgumentException("Key " +
                            key + " does not exist in map.");  }
   return values[findKey(key)];
}
```

```
$ java ExceptionDemo
Exception in thread "main"
java.lang.IllegalArgumentException: Key yolp does not
exist in map.
    at ArrayMap.get(ArrayMap.java:40)
    at ExceptionDemo.main(ExceptionDemo.java:6)
```

Handling Errors

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: -1
    at ArrayMap.get(ArrayMap.java:38)
```

Sometimes things go wrong, e.g.

- You try to use 383,124 gigabytes of memory.
- You try to cast an Object as a Dog, but dynamic type is not Dog.
- You try to call a method using a reference variable that is equal to null.
- You try to access index -1 of an array.

The Java approach to handling these exceptional events is to ***throw*** an ***exception***.

- Disrupts normal flow of the program.
- So far in 61B, exceptions just cause the program to crash, printing out a helpful (?) message for the user.

Exceptions: May be Explicitly or Implicitly Thrown

Java itself can throw exceptions implicitly, e.g.

```
Object o = "mulchor";
Planet x = (Planet) o;
```

```
Exception in thread "main" java.
lang.ClassCastException:
java.lang.String cannot be cast to
Planet
```

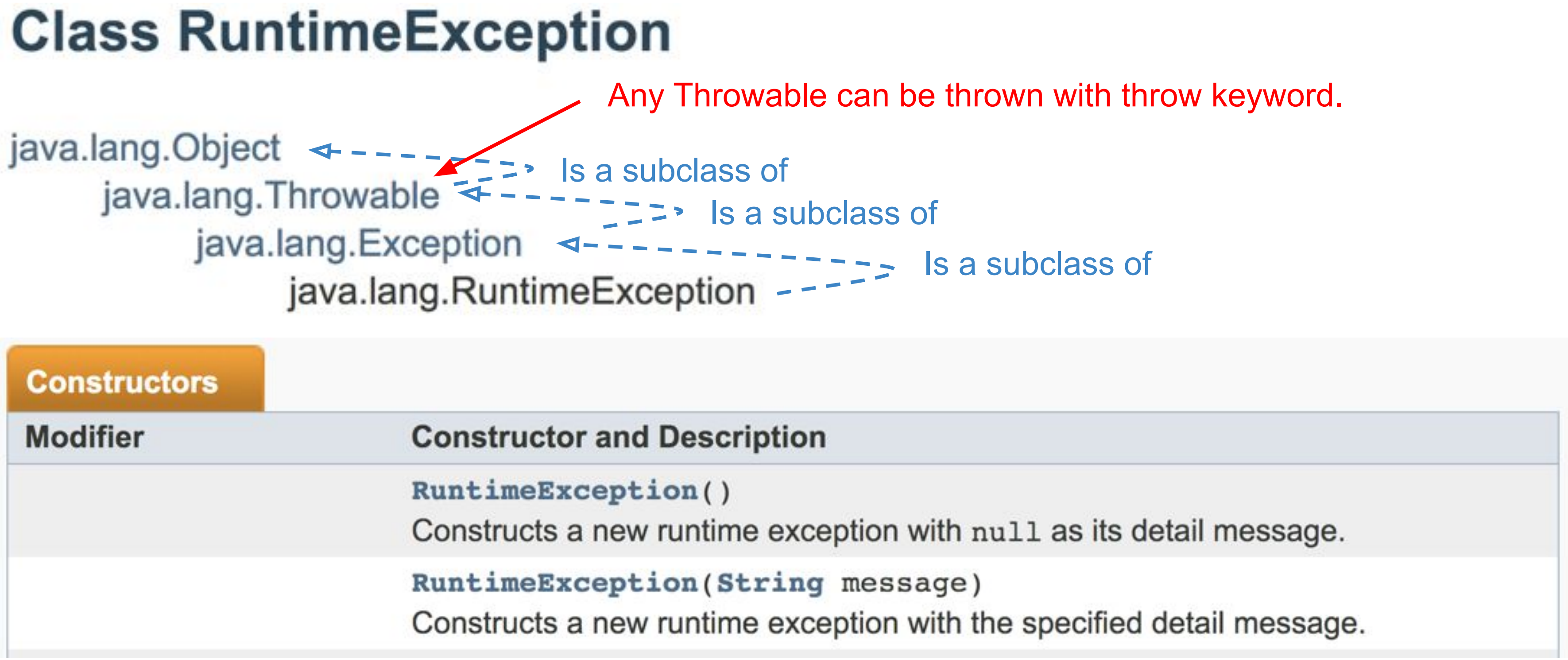
Java code can also throw exceptions explicitly using ***throw*** keyword:

```
public static void main(String[] args) {
    System.out.println("ayyy lmao");
    throw new RuntimeException("For no reason.");
}
```

Creates new object of type RuntimeException!

```
$ java Alien
ayyy lmao
Exception in thread "main"
java.lang.RuntimeException: For no
reason.
        at Alien.main(Alien.java:4)
```

RuntimeException API



Class RuntimeException

Any Throwable can be thrown with throw keyword.

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
```

Is a subclass of

Is a subclass of

Is a subclass of

Constructors

Modifier	Constructor and Description
	`RuntimeException()` Constructs a new runtime exception with `null` as its detail message.
	`RuntimeException(String message)` Constructs a new runtime exception with the specified detail message.

Exceptions are instances of classes like most everything else in Java.

What has been Thrown, can be Caught

So far, thrown exceptions cause code to crash.

- Can ‘catch’ exceptions instead, preventing program from crashing. Use keywords ***try*** and ***catch*** to break normal flow.

```
Dog d = new Dog("Lucy", "Retriever", 80);
d.becomeAngry();

try {
    d.receivePat();
} catch (Exception e) {
    System.out.println(
       "Tried to pat: " + e);
}

System.out.println(d);
```

```
$ java ExceptionDemo
Tried to pat: java.lang.RuntimeException:
grrr... snarl snarl
Lucy is a displeased Retriever weighing
80.0 standard lb units.
```

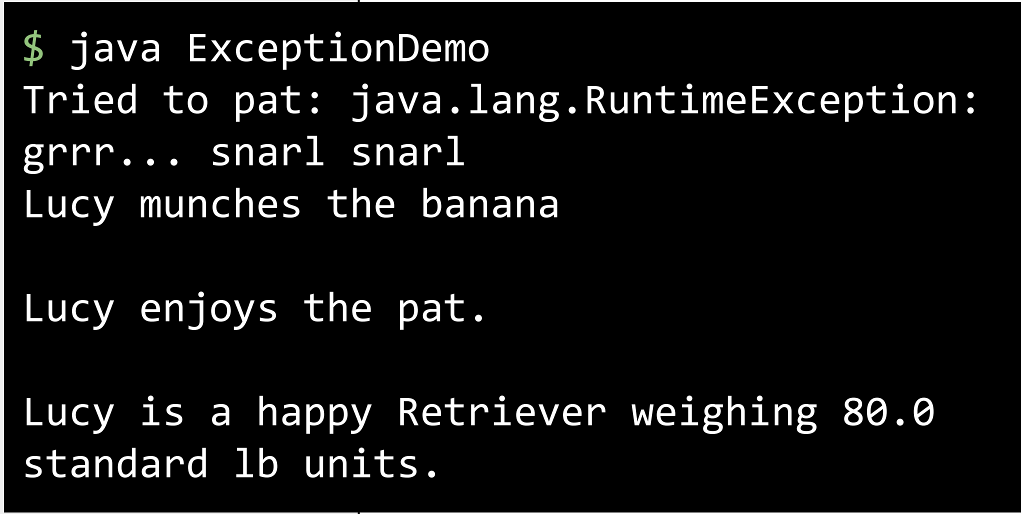
Code does not crash since we caught the RuntimeException thrown by the dog.

Can take Corrective Action in Catch Blocks

Catch blocks can execute arbitrary code.

- May include corrective action.

```java
Dog d = new Dog("Lucy", "Retriever", 80);
d.becomeAngry();

try {
    d.receivePat();
} catch (Exception e) {
    System.out.println(
        "Tried to pat: " + e);
    d.eatTreat("banana");
}
d.receivePat();
System.out.println(d);
```

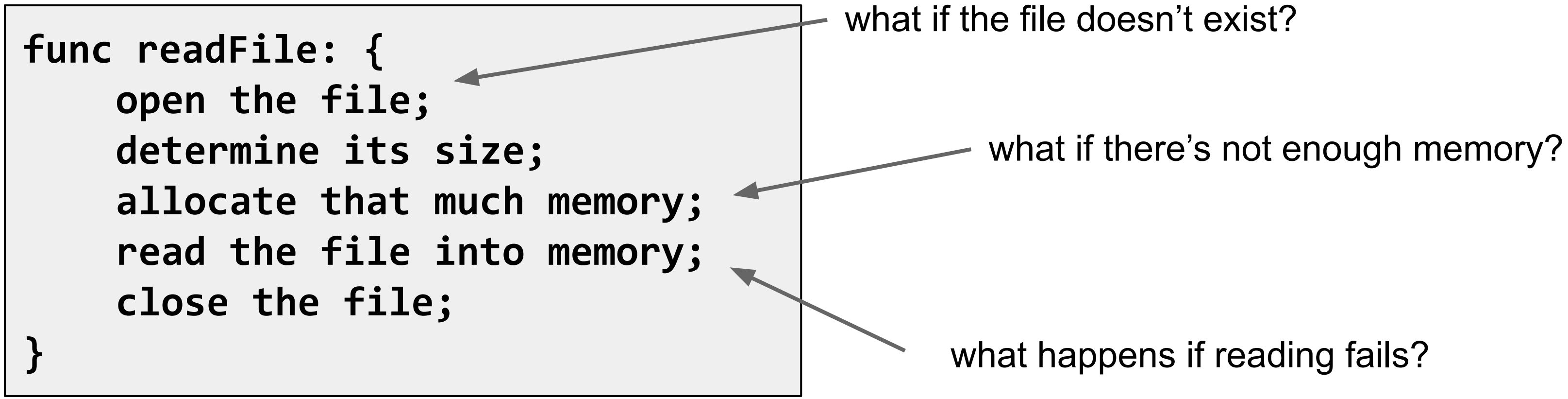


```
$ java ExceptionDemo
Tried to pat: java.lang.RuntimeException:
grrr... snarl snarl
Lucy munches the banana

Lucy enjoys the pat.

Lucy is a happy Retriever weighing 80.0
standard lb units.
```

Why Exceptions?

Allows you to keep error handling code separate from ‘real’ code.

- Consider pseudocode that reads a file:



```
func readFile: {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

what if the file doesn’t exist?

what if there’s not enough memory?

what happens if reading fails?

Error Handling Code (Naive)

One naive approach to the right.

- Clearly a bad idea.

```
func readFile: {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

```
func readFile: {
  open the file;
  if (theFileIsOpen) {
    determine its size;
    if (gotTheFileLength) {
      allocate that much memory;
     } else {
       return error("fileLengthError");
     }
     if (gotEnoughMemory) {
      read the file into memory;
      if (readFailed) {
       return error("readError");
      }
      ...
    } else {
     return error("memoryError");
    }
   } else {
    return error("fileOpenError")
  }
}
```

With Exceptions

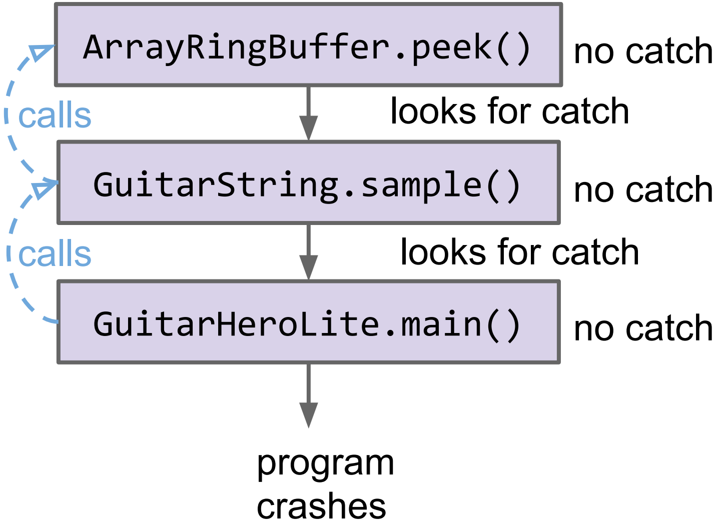
```
func readFile: {
    try {
       open the file;
       determine its size;
       allocate that much memory;
       read the file into memory;
       close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
       doSomething;
    } catch (memoryAllocationFailed) {
       doSomething;
    } catch (readFailed) {
       doSomething;
    } catch (fileCloseFailed) {
       doSomething;
    }
}
```

```
func readFile: {
    open the file;
    if (theFileIsOpen) {
        determine its size;
        if (gotTheFileLength) {
           allocate that much memory;
         } else {
              return error("fileLengthError");
         }
         if (gotEnoughMemory) {
           read the file into memory;
            if (readFailed) {
              return error("readError");
            }
           ...
        } else {
          return error("memoryError");
        }
     } else {
        return error("fileOpenError")
     }
```

Exceptions and the Call Stack

When an exception is thrown, it descends the call stack.

- If exceptions reaches the bottom of the stack, the program crashes and Java provides a message for the user.
  - Ideally the user is a programmer with the power to do something about it.



```
java.lang.RuntimeException in thread “main”:
    at ArrayRingBuffer.peek:63
    at GuitarString.sample:48
    at GuitarHeroLite.java:110
```

“Must be Caught or Declared to be Thrown”

Occasionally, you’ll find that your code won’t even compile, for the mysterious reason that an exception “must be caught or declared to be thrown”.

- The basic idea: Some exceptions are considered so disgusting by the compiler that you MUST handle them somehow.
- We call these “checked” exceptions.

“Must be checked exceptions” is a more accurate name.

```java
public static void main(String[] args) {
    Eagle.gulgate();
}
```

```
$ javac What.java
What.java:2: error: unreported exception IOException; must be caught or
declared to be thrown
        Eagle.gulgate();
        ^
```

Checked Exceptions

Examples so far have been ***unchecked*** exceptions. There are also ***checked*** exceptions:

To be defined soon...

- Compiler requires that these be “caught” or “specified”.
  - Goal: Disallow compilation to prevent avoidable program crashes.
- Example:

```
public class Eagle {
   public static void gulgate() {
      if (today == “Thursday”) {
         throw new IOException("hi"); }
   }
}
```

```
$ javac Eagle
Eagle.java:4: error: unreported exception IOException; must be caught
or declared to be thrown
        throw new IOException("hi"); }
        ^
```

Unchecked Exceptions

By contrast unchecked exceptions have no such restrictions.

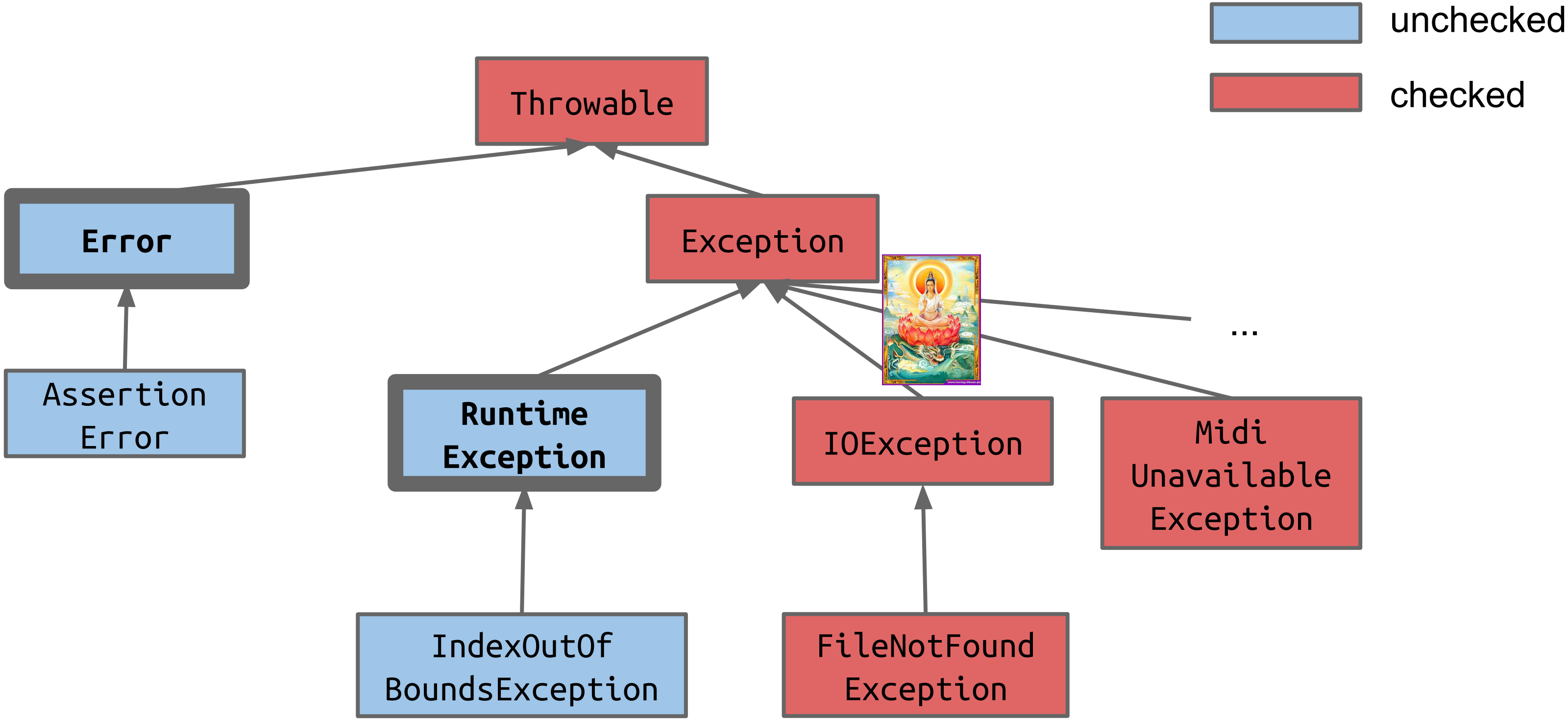
- Code below will compile just fine (but will crash at runtime).

```java
public class UncheckedExceptionDemo {
    public static void main(String[] args) {
        if (today == “Thursday”) {
            throw new RuntimeException("as a joke"); }
    }
}
```

```
$ javac UncheckedExceptionDemo.java
$ java UncheckedExceptionDemo
Exception in thread "main" java.lang.RuntimeException: as a joke.
    at UncheckedExceptionDemo.main(UncheckedExceptionDemo.java:3)
```

Checked vs. Unchecked Exceptions

Any subclass of **RuntimeException** or **Error** is *unchecked*, all other Throwables are *checked*.



Checked vs. Unchecked

- Compiles fine, because the possibility of unchecked exceptions is allowed:

```
public class UncheckedExceptionDemo {
    public static void main(String[] args) {
       if (today == “Thursday”) {
          throw new RuntimeException("as a joke"); }
    }
}
```

Java considers this an “unchecked” exception.

Why didn’t you catch or specify??

- Won’t compile, because there exists possibility of checked exception.

```
public class Eagle {
    public static void gulgate() {
       if (today == “Thursday”) {
          throw new IOException("hi"); }
    }
}
```

Java considers this a “checked” exception.

Checking Exceptions

Compiler requires that all checked exceptions be **caught** or **specified**.

Two ways to satisfy compiler:

- **Catch**: Use a catch block after potential exception.

```
public static void gulgate() {
    try {
        if (today == “Thursday”) {
          throw new IOException("hi"); }
    } catch (Exception e) {
       System.out.println("psych!");
    }
}
```

- **Specify** method as dangerous with ***throws*** keyword.

Checking Exceptions

Compiler requires that all checked exceptions be **caught** or **specified**.

Two ways to satisfy compiler:

- **Catch**: Use a catch block after potential exception.
- **Specify** method as dangerous with ***throws*** keyword.
  - Defers to someone else to handle exception.

```
public static void gulgate() throws IOException {
   ... throw new IOException("hi"); ...
}
```

Checking Exceptions

If a method uses a 'dangerous' method (i.e. might throw a checked exception), it becomes dangerous itself.

```
public static void gulgate() throws IOException {
   ... throw new IOException("hi"); ...
}
```

```
public static void main(String[] args) {
    Eagle.gulgate();
}
```

> *"He who fights with monsters should look to it that he himself does not become a monster. And when you gaze long into an abyss the abyss also gazes into you." - Beyond Good and Evil (Nietzsche)*

How do we fix this?

Catch or specify!

Checking Exceptions

Two ways to satisfy compiler: *Catch* or *specify* exception.

```
public static void gulgate() throws IOException {
   ... throw new IOException("hi"); ...
}
```

```
public static void main(String[] args) {
  try {
    gulgate();
  } catch(IOException e) {
    System.out.println("Averted!");
  }
}
```

Catch an Exception:
Keeps it from getting out.

Use when you can handle the problem.

```
public static void main(String[] args)
    throws IOException {
      gulgate();
}
```

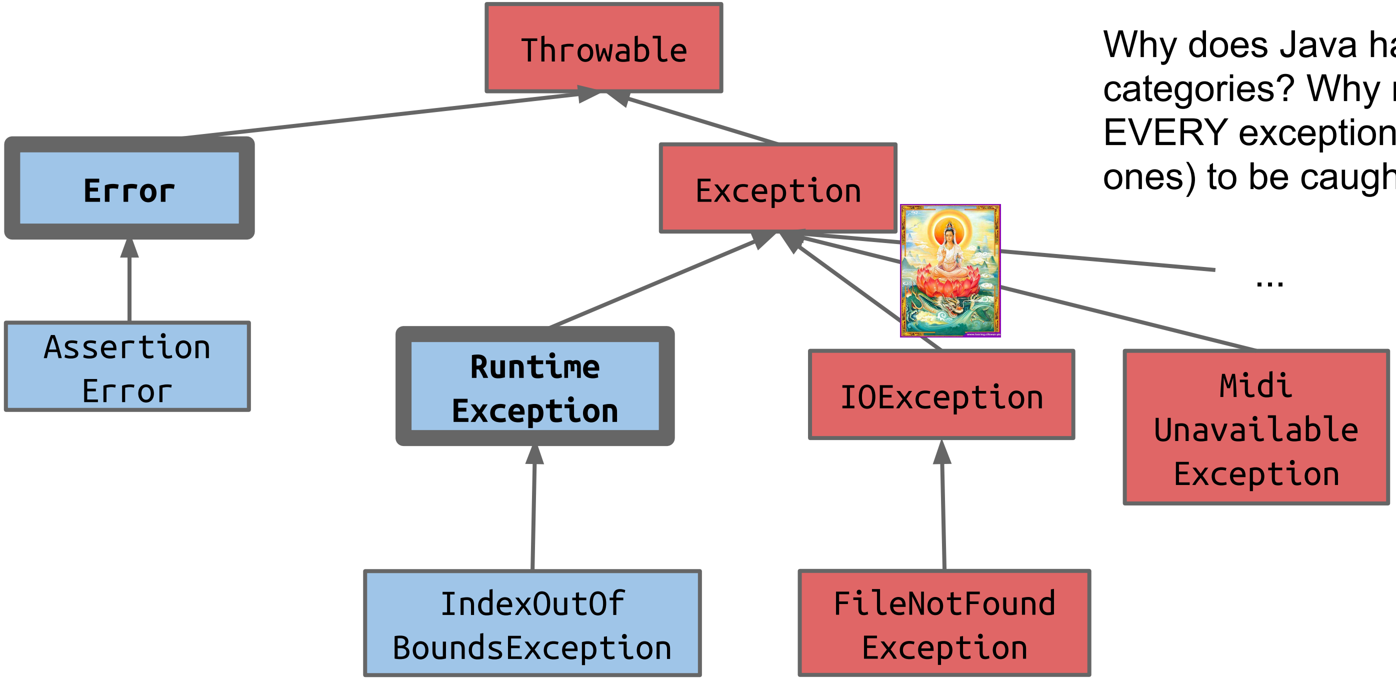
Specify that you might throw an exception.

Use when someone else should handle.

Checked vs. Unchecked Exceptions

Checked exceptions are part of the **specification** of a class.

- If you look up the nice documentation it will appear: (example).



Why does Java have two categories? Why not require EVERY exception (even blue ones) to be caught or declared?

Iteration

The Enhanced For Loop

We saw that Java allows us to iterate through Lists using a convenient shorthand syntax sometimes called the “foreach” or “enhanced for” loop.

- Let’s strip away the magic so we can build our own classes that support this.

```
List<Integer> friends =
  new ArrayList<Integer>();
friends.add(5);
friends.add(23);
friends.add(42);
for (int x : friends) {
   System.out.println(x);
}
```

Doing Things The Hard Way

An alternate, uglier way to iterate through a List is to use the iterator() method.

List.java:

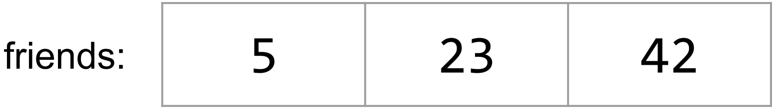
```
public Iterator<E> iterator();
```

```
List<Integer> friends =
  new ArrayList<Integer>();
...
for (int x : friends) {
   System.out.println(x);
}
```

```
List<Integer> friends =
  new ArrayList<Integer>();
...
Iterator<Integer> seer
     = friends.iterator();

while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the iterator() method.



friends: 5 | 23 | 42

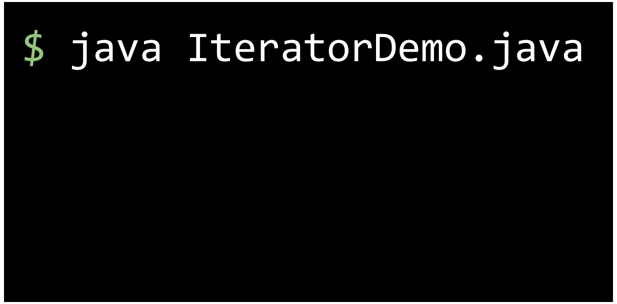
```
Iterator<Integer> seer
     = friends.iterator();
while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the iterator() method.

friends:

5	23	42



```
$ java IteratorDemo.java
```

```
Iterator<Integer> seer
     = friends.iterator();
while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the iterator() method.

friends:			
	5	23	42

```
$ java IteratorDemo.java
```

```java
Iterator<Integer> seer
     = friends.iterator();
while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the iterator() method.

friends:	5	23	42

```
$ java IteratorDemo.java
5
```

```
Iterator<Integer> seer
     = friends.iterator();
while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the iterator() method.

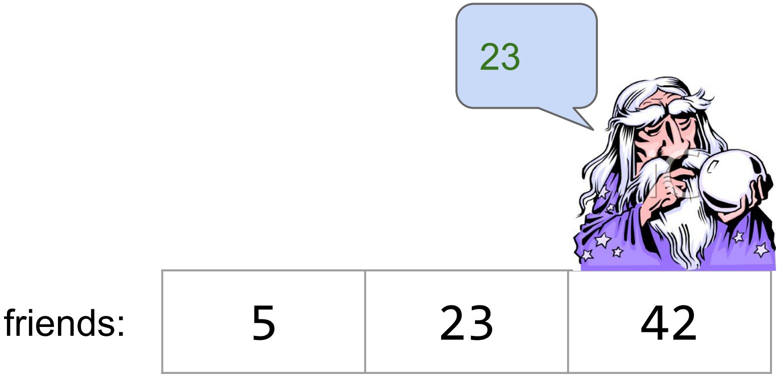
friends:	5	23	42

```
$ java IteratorDemo.java
5
```

```
Iterator<Integer> seer
     = friends.iterator();
while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the iterator() method.



```
$ java IteratorDemo.java
5
23
```

```
Iterator<Integer> seer
     = friends.iterator();
while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the iterator() method.

friends:

5	23	42

```
$ java IteratorDemo.java
5
23
```

```
Iterator<Integer> seer
     = friends.iterator();
while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the iterator() method.

```
$ java IteratorDemo.java
5
23
42
```

friends:

5	23	42

```
Iterator<Integer> seer
     = friends.iterator();
while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

How Iterators Work

An alternate, uglier way to iterate through a List is to use the iterator() method.

friends:	5	23	42

```
$ java IteratorDemo.java
5
23
42
```

```
Iterator<Integer> seer
     = friends.iterator();
while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

The Secret of the Enhanced For Loop

The secret: The code on the left is just shorthand for the code on the right. For code on right to work, which checks does the compiler need to do?

A. Does the List interface have an iterator() method?
B. Does the List interface have next/hasNext() methods?
C. Does the Iterator interface have an iterator method?
D. Does the Iterator interface have next/hasNext() methods?

```
List<Integer> friends = new ArrayList<Integer>();
```

```
for (int x : friends) {
   System.out.println(x);
}
```

```
Iterator<Integer> seer
     = friends.iterator();

while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

The Secret of the Enhanced For Loop

For code on the right to work:

- Compiler checks that Lists have a method called iterator() that returns an Iterator<Integer>.
- Then, compiler checks that Iterators have:
  - hasNext()
  - next()

```
List<Integer> friends = new ArrayList<Integer>();
```

```
for (int x : friends) {
   System.out.println(x);
}
```

```
Iterator<Integer> seer
     = friends.iterator();

while (seer.hasNext()) {
  System.out.println(seer.next());
}
```

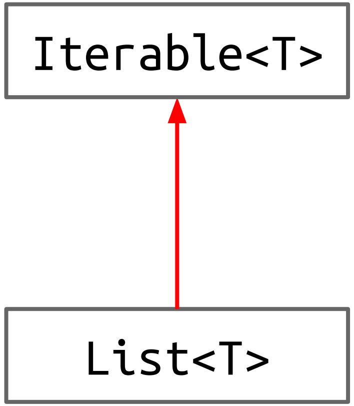
The Iterable Interface

Compiler checks that Lists have a method called iterator() that returns an Iterator<Integer>.

- **How**: The List interface extends the Iterable interface, inheriting the abstract iterator() method*.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public interface List<T> extends Iterable<T>{
    ...
}
```



*: Actually List extends Collection which extends Iterable, but this is close enough to the truth. Also I'm omitting some default methods in the Iterable interface.

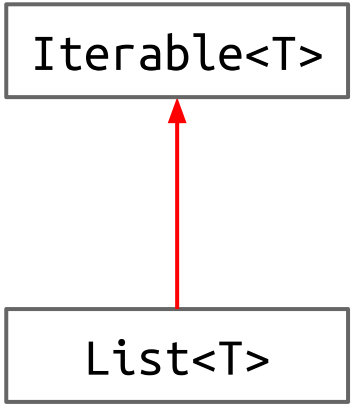
The Iterator Interface

Then, compiler checks that Iterators have hasNext and next()

- **How**: The Iterator interface specifies these abstract methods explicitly.

Iterator<T>

```
package java.util;
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```



I'm omitting some default methods in the Iterator interface.

Iteration Using A Nested Class

A “client program" is just any program that uses our class.

First, let’s create a KeyIterator class that allows client programs to iterate through the keys of an ArrayMap, as well as simple client program.

- See the study guide for this lecture for starter code (ArrayMap.java and IterationDemo.java).

Iteration Using A Nested Class

First, let's create a KeyIterator class that allows client programs to iterate through the keys of an ArrayMap.

```
public class KeyIterator {
    private int ptr;
    public KeyIterator() {
        ptr = 0;
    }
    public boolean hasNext() {
       return (ptr != size);
    }
    public K next() {
        K returnItem = keys[ptr];
        ptr = ptr + 1;
        return returnItem;
   }
}
```

ArrayMap.java

```
ArrayMap<String, Integer> am =
  new ArrayMap<String, Integer>();
am.put("hello", 5);
am.put("syrups", 10);
ArrayMap.KeyIterator ami =
   am.new KeyIterator();

while (ami.hasNext()) {
   System.out.println(ami.next());
}
```

Instantiating nested classes requires dot notation.

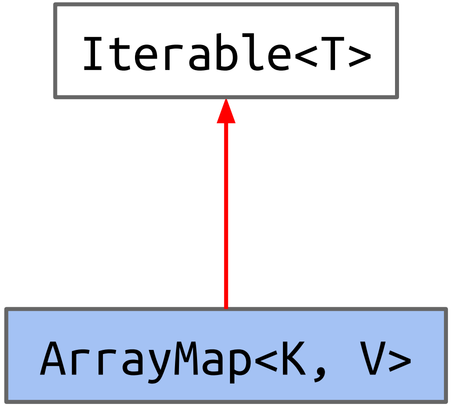
IterationDemo.java

For-each Iteration And ArrayMaps

To support the enhanced for loop, we need to make ArrayMap implement the Iterable interface.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public class ArrayMap<K, V> {
    ...
}
```

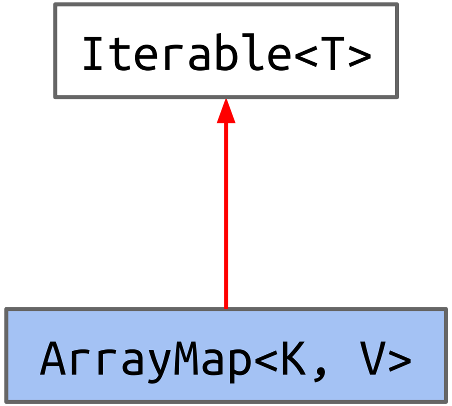


For-each Iteration And ArrayMaps

To support the enhanced for loop, we need to make ArrayMap implement the Iterable interface.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public class ArrayMap<K, V> implements Iterable<K>
    @Override
    public Iterator<T> iterator() {
        return new KeyIterator();
    }
}
```



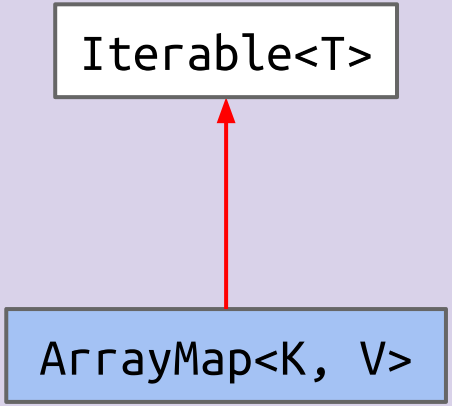
For-each Iteration And ArrayMaps

Given our definition of KeyIterator earlier, the code below will not compile.

- Why?

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public class ArrayMap<K, V> implements Iterable<K>
    @Override
    public Iterator<T> iterator() {
        return new KeyIterator();
    }
}
```



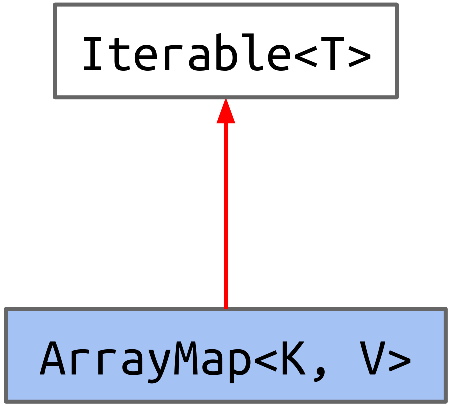
For-each Iteration And ArrayMaps

Given our definition of KeyIterator earlier, the code below will not compile.

- KeyIterator does not implement the Iterator interface.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public class ArrayMap<K, V> implements Iterable<K>
    @Override
    public Iterator<T> iterator() {
       return new KeyIterator();
    }
}
```

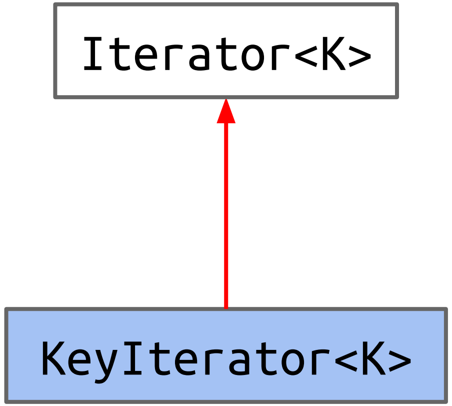


For-each Iteration And ArrayMaps

To complete our task, simply make KeyIterator extend Iterator.

```
package java.util;
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

```
public class KeyIterator implements Iterator<K> {
   private int ptr;
   public KeyIterator() { ptr = 0; }
   public boolean hasNext() { return (ptr != size); }
   public K next() { ... }
}
```



Iterator<K>

KeyIterator<K>

Iteration Summary

Implement iterable interface to support enhanced for loop.

- iterator() method must return an object that implements the Iterator interface.

Part 5 of HW1 gives you a chance to try this out yourself.

Citations

Seer:
http://www.clipartoday.com/_thumbs/022/Fantasy/astrology_crystal_190660_tnb.png

Exceptions guy (why does this image exist, IDK):
http://education.oge.gov/training/module_files/ogewrkctr_wbt_07/exception.jpg