Introduction to Object Oriented Programming

Roy Schwartz, The Hebrew University (67125)

Lecture 7:

Core Java Topics

Last Week

- Intro to Generics
- Java Collections

Lecture 7a: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

Lecture 7a: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

- Compilation errors
  - Detected by the compiler

- Compilation errors
  - Detected by the compiler
- Runtime errors
  - **Not** detected by the compiler
  - Require error handling
  - Result of:
    - Bugs
    - Bad input
    - …

- Compilation errors
  - Detected by the compiler
- Runtime errors
  - **Not** detected by the compiler
  - Require error handling
  - Result of:
    - Bugs
    - Bad input
    - …

Runtime Error Handling

- Runtime error handling is a major issue in programming

Runtime Error Handling

- Runtime error handling is a major issue in programming
- A good program:
  - Can recover from errors whenever it is possible
    - Its documentation must provide information about the cases when it isn't possible

Runtime Error Handling

- Runtime error handling is a major issue in programming
- A good program:
  - Can recover from errors whenever it is possible
    - Its documentation must provide information about the cases when it isn't possible
  - Handles errors in the most appropriate place
    - Not too early, not too late

Exceptions

- An ***exception*** is a **message** that states that something went wrong
  - Alternative to return values

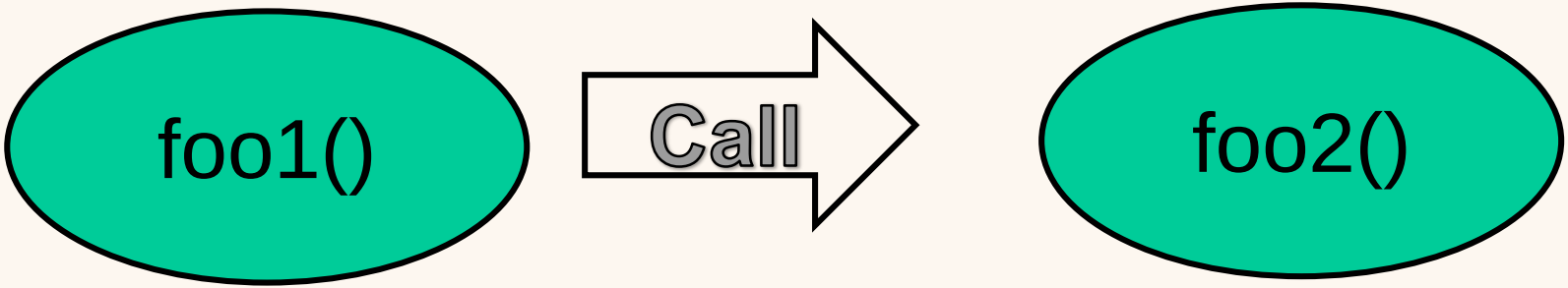
Exceptions

- An ***exception*** is a **message** that states that something went wrong
  - Alternative to return values
- When there is a problem and some method cannot continue to run properly, this message is passed back to the calling method
  - This method can decide whether and how it is able to handle this error
  - If it cannot handle it, it will send another **message** to its calling method

Exceptions

- An ***exception*** is a **message** that states that something went wrong
  - Alternative to return values
- When there is a problem and some method cannot continue to run properly, this message is passed back to the calling method
  - This method can decide whether and how it is able to handle this error
  - If it cannot handle it, it will send another **message** to its calling method
- In java, exceptions are **Objects**

Exceptions



Exceptions

Exceptions

Throwing Exceptions

```
int get(int index) throws ListException{
                        if (list.isEmpty()) {
                                throw new ListException();
                        }
                        //...
                }
}
```

Throwing Exceptions

```
int get(int index) throws ListException{
                        if (list.isEmpty()) {
                                throw new ListException();
                        }
                        //...
                }
}
```

Throwing Exceptions

```
int get(int index) throws ListException{
                        if (list.isEmpty()) {
                                throw new ListException();
                        }
                        //...
                }
}
```

Throwing Exceptions

throw new ListException();

- Called whenever an error is detected

Throwing Exceptions

throw new ListException();

- Called whenever an error is detected
- Results in terminating the method immediately
  - No other code runs
  - No value is returned

Specifying Exceptions

int get(**int** index) **throws** ListException

- Indicates that this method **can potentially throw** this exception

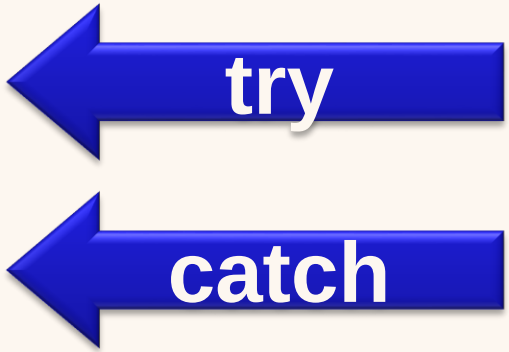
Specifying Exceptions

int get(**int** index) **throws** ListException

- Indicates that this method **can potentially throw** this exception
- Part of the method declaration
  - Should be documented in the API
  - Use the tag @throws in javadoc

Handling Exceptions

```
try {
        //get index from user
        int element = list.get(0);
        //...
} catch(ListException e) {
        // handle list errors
} catch(OtherException e) {
        // handle other errors
}
// Rest of program
```



Catching Exceptions

catch(ListException e)

- Code is executed if one of the methods in the **try** block throws a ListException

Catching Exceptions

catch(ListException e)

- Code is executed if one of the methods in the **try** block throws a ListException
- ListException e:
  - Catch the specific exception object (object named *e* of type ListException)
  - This object may contain useful information about the error that caused this exception

Throwing Back an Exception

- A method that cannot handle an error can pass it back to its caller

```
public void foo() throws ListException() {
        //get index from user
        int element = list.get(0);
        …
}
```

Throwing Back an Exception

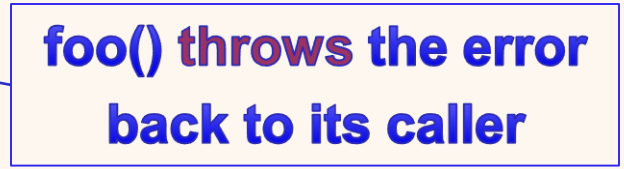
- A method that cannot handle an error can pass it back to its caller

```
public void foo() throws ListException() {
        //get index from user
        int element = list.get(0);
        …
}
```

Throwing Back an Exception

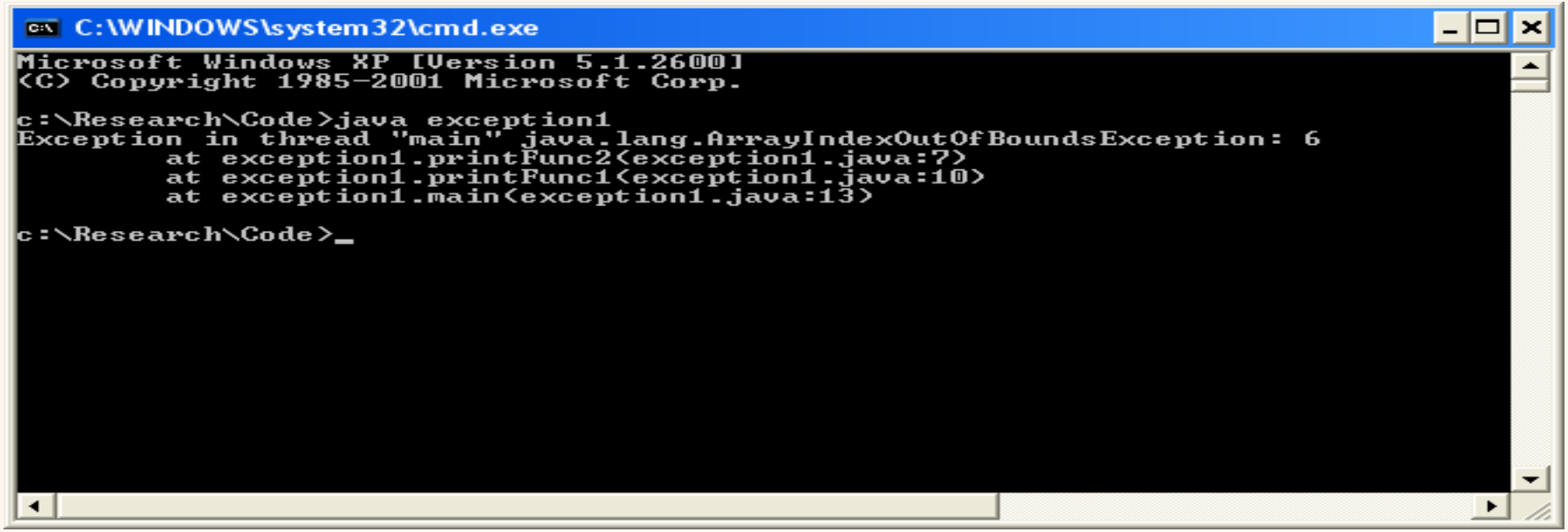
- A method that cannot handle an error can pass it back to its caller

```
public void foo() throws ListException() {
        //get index from user
        int element = list.get(0);
        …
}
```



Some Old Memories…

Exception “Not handled”



Lecture 7b: Overview

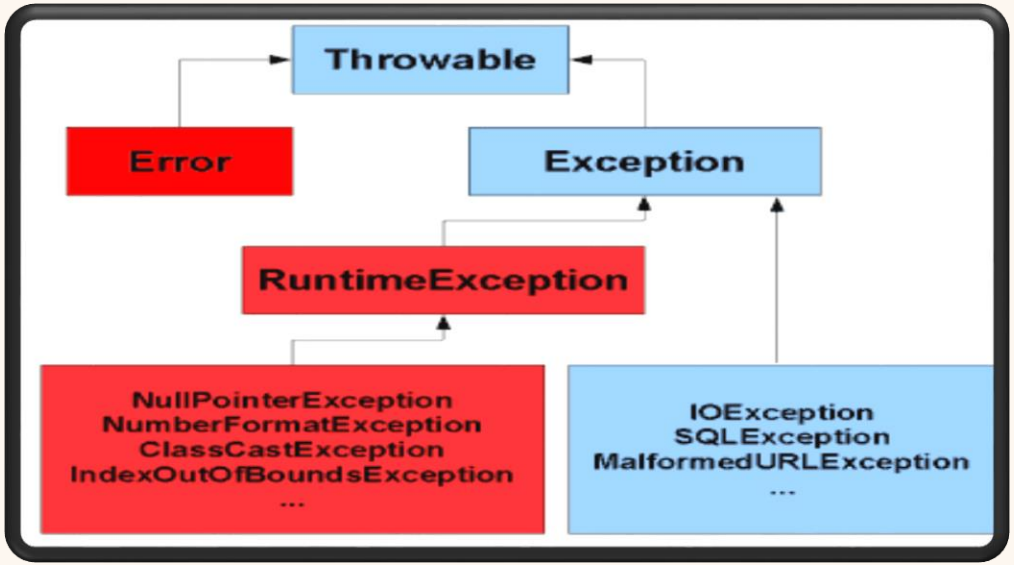
- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

Lecture 7b: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

Hierarchy:

Exceptions are objects

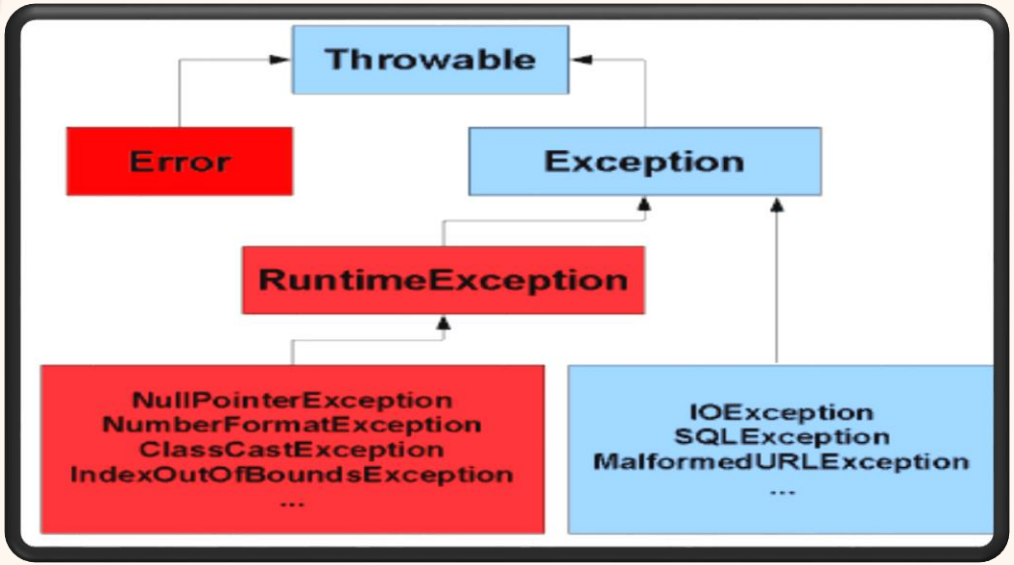


http://www.javamex.com/tutorials/exceptions/exceptions_hierarchy.shtml

Hierarchy:

Exceptions are objects

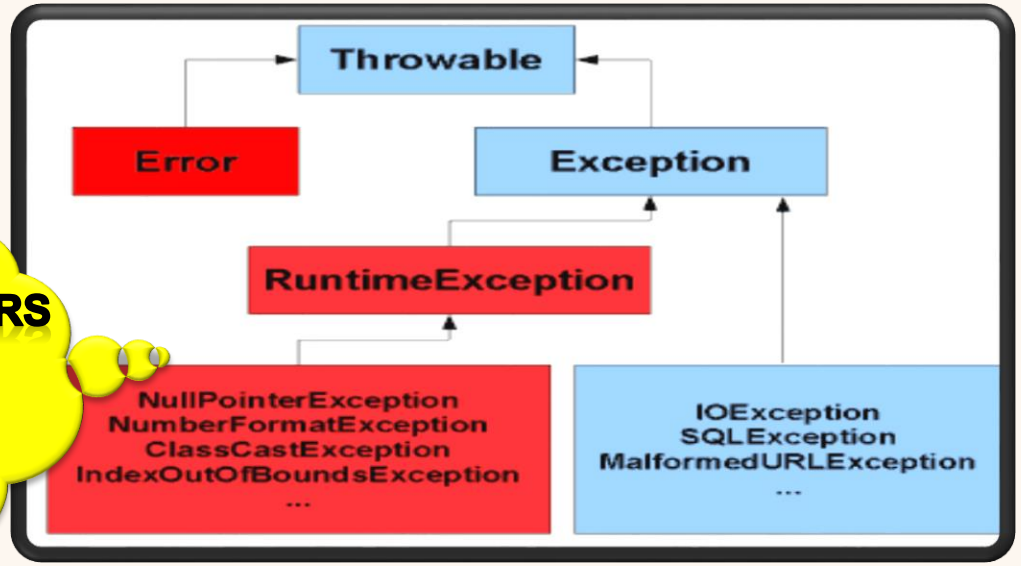
SYSTEM ERRORS
Shouldn't expect to catch and recover from



http://www.javamex.com/tutorials/exceptions/exceptions_hierarchy.shtml

Hierarchy:

Exceptions are objects



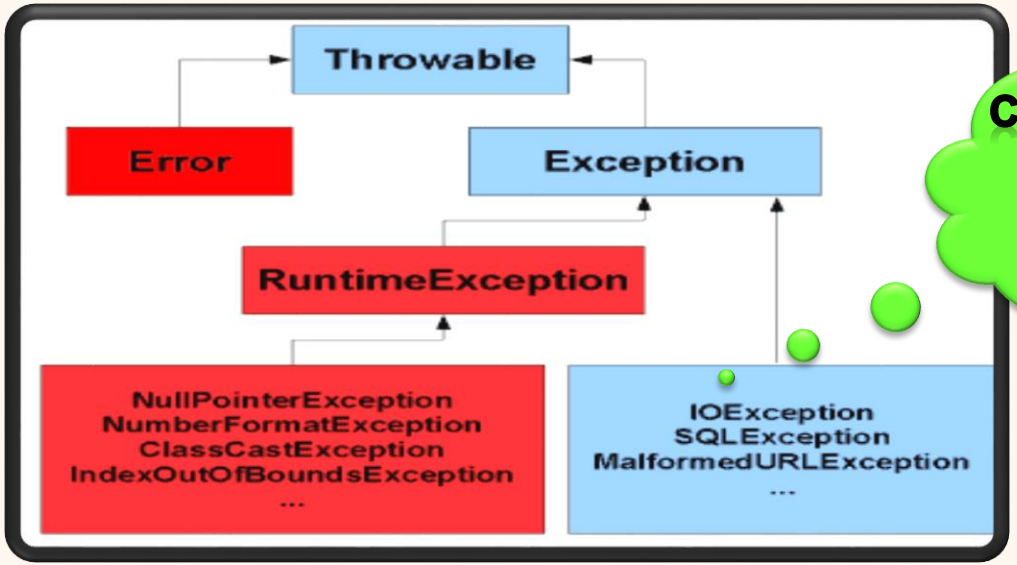
UNCHECKED ERRORS

Shouldn't generally expect to occur, but could potentially recover from

http://www.javamex.com/tutorials/exceptions/exceptions_hierarchy.shtml

Hierarchy:

Exceptions are objects



CHECKED ERRORS

Expect to occur in the normal course of duty

http://www.javamex.com/tutorials/exceptions/exceptions_hierarchy.shtml

Throwable Types

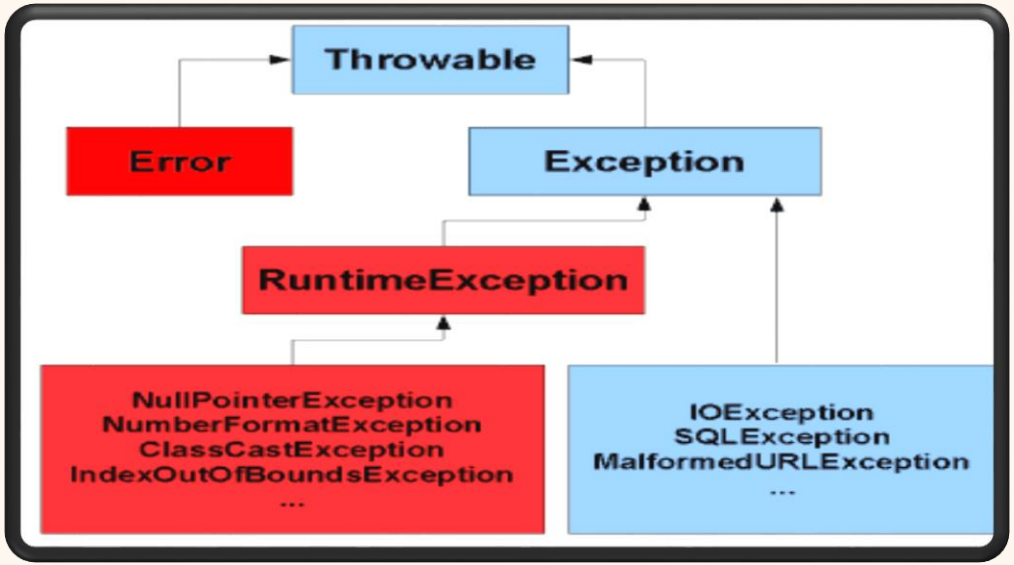
- **Checked**
  - Extend the **Exception** class
  - Usually result from *user* errors (wrong file name, bad URL, …)
    - IOException, MalformedURLException, etc.
  - Often specific to one program
  - Checked exceptions must **both**
    - appear in the **throws** statement **and**
    - be handled by the calling method (using a **try**/**catch** clause, or **throw**ing them back)
  - Not doing so results in **a compilation error**

Throwable Types

- **Unchecked**
  - Extend **RuntimeException**
  - Usually result from *programming* errors (null pointer, index out of bounds, division by zero, etc.)
    - ArrayIndexOutOfBoundsException, NullPointerException, etc.
  - Could occur in many different programs and scenarios
  - No need for the **throws** statement, nor handling by the calling method
    - For better documentation, specify non-trivial exceptions

Hierarchy:

Exceptions are objects



http://www.javamex.com/tutorials/exceptions/exceptions_hierarchy.shtml

Lecture 7c: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

Lecture 7c: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

Why Exceptions?

- **Separating** error handling code from the rest of the code
- **Error propagating** up the call stack
- **Grouping** together and **differentiating** error types

Separating Error Handling Code

- Consider the following operations on some data

```
readData {
    ask for data size;
    allocate required memory;
    read the data from user into memory;
}
```

- Without exceptions, error handling will be “on the fly”

Without Exceptions

```
public int readData {
        if (!ask the data size)
                return -1;
        if (!allocate memory)
                return -2;
        if (!read data)
                return -3;

        return 0;
}
```

Without Exceptions

```
public int readData {
    if (!ask the data size)
        return -1;
    if (!allocate memory)
        return -2;
    if (!read data)
        return -3;

    return 0;
}
```

With Exceptions

```
public void readData {
        try {
                ask for data size;
                allocate required memory;
                read the data from user into memory;
         }
         catch (DataSizeException e) { doSomething; }
         catch (OutOfMemException e) { doSomething; }
         catch (ReadDataException e) { doSomething; }
}
```

Error Propagating up the Call Stack

- Consider the following setting:

$foo_1()$ ➔ $foo_2()$ ➔ … $foo_n()$ ➔ **error**!

Error Propagating up the Call Stack

- Consider the following setting:

$foo_1() \rightarrow foo_2() \rightarrow \ldots foo_n() \rightarrow$ **error**!

- The only method that can recover from this error is $foo_1()$
  - All other foo_i methods have nothing to do but pass the error back to $foo_{i-1}()$

Error Propagating up the Call Stack

- Without Exceptions:

Error Propagating up the Call Stack

- Without Exceptions:

```
public boolean foo_n() {
    // Some foo_n() code
    if (error)
        return false;
    // Some more foo_n() code
    return true;
}
```

Error Propagating up the Call Stack

- Without Exceptions:

```
public boolean foo_n() {
    // Some foo_n() code
    if (error)
        return false;
    // Some more foo_n() code
    return true;
}
```

```
foreach 1 <= i < n:
public boolean foo_i() {
    // Some foo_i() code
    if (!foo_{i+1}())
        return false;
    // Some more foo_i() code
    return true;
}
```

Error Propagating up the Call Stack

- Without Exceptions:

```
public boolean foo_n() {
    // Some foo_n() code
    if (error)
        return false;
    // Some more foo_n() code
    return true;
}
```

```
foreach 1 <= i < n:
public boolean foo_i() {
    // Some foo_i() code
    if (!foo_{i+1}())
        return false;
    // Some more foo_i() code
    return true;
}
```

- With Exceptions:

```
foreach 1 <= i < n:
public void foo_n throws SomeException()  {
    // Some foo_n() code
    if (error)
        throw new SomeException();
    // Some more foo_n() code
}
public void foo_i throws SomeException()  {
    // Some foo_i() code
    foo_{i+1}();
    // Some more foo_i() code
 }
```

Grouping Together and Differentiating Error Types

```
class ListException extends Exception{…}

class EmptyListException extends ListException {…}
class InvalidIndexException extends ListException {…}

long get(int index) throws ListException{
      if (list.isEmpty())
            throw new EmptyListException();
      if (list.size() <=  index)
            throw new InvalidIndexException();
      //...
}
```

Grouping Together and Differentiating Error Types

```
class ListException extends Exception{…}

class EmptyListException extends ListException {…}
class InvalidIndexException extends ListException {…}

long get(int index) throws ListException{
        if (list.isEmpty())
                throw new EmptyListException();
        if (list.size() <=  index)
                throw new InvalidIndexException();
        //…
}
```

```
try {
        long l = myList.get(5);
} catch (EmptyListException e) {
        // handle empty lists
} catch (InvalidIndexException e) {
        // handle invalid index
}
```

Grouping Together and Differentiating Error Types

```
class ListException extends Exception{…}

class EmptyListException extends ListException {…}
class InvalidIndexException extends ListException {…}

long get(int index) throws ListException{
        if (list.isEmpty())
                throw new EmptyListException();
        if (list.size() <=  index)
                throw new InvalidIndexException();
        //…
}
```

```
try {
        long l = myList.get(5);
} catch (EmptyListException e) {
        // handle empty lists
} catch (InvalidIndexException e) {
        // handle invalid index
}
```

```
try {
        long l = myList.get(5);
} catch (ListException e) {
        // handle any list error
}
```

Design ANTI-Pattern

Expection Handling

- Using a computer language's error handling structures to perform normal program logic

```
for (int i = 0 ; i < prodnums.length ; i++) {
      displayProductInfo(prodnums[i]);
}
// Do some cleanup
```

Design ANTI-Pattern

Expection Handling

- Using a computer language's error handling structures to perform normal program logic

```
for (int i = 0 ; i < prodnums.length ; i++) {
      displayProductInfo(prodnums[i]);
}
// Do some cleanup
```

```
try {
      int idx = 0;
      while (true) {
            displayProductInfo(prodnums[idx]);
            idx++;
      }
 } catch (IndexOutOfBoundsException ex) {
      // Do some cleanup
}
```

Design ANTI-Pattern

Expection Handling

- Using a computer language's error handling structures to perform normal program logic

```
for (int i = 0 ; i < prodnums.length ; i++) {
      displayProductInfo(prodnums[i]);
}
// Do some cleanup
```

✘Heavy!
✘Unexpected!
✘Hides bugs!

```
try {
      int idx = 0;
      while (true) {
            displayProductInfo(prodnums[idx]);
            idx++;
      }
 } catch (IndexOutOfBoundsException ex) {
      // Do some cleanup
}
```

Uncommon Usage of Exceptions

- Returning data from deep calls

- Timeout

- UnsupportedOperationException

Reminder: Collection<E> Interface

```
public interface Collection<E> extends
                Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    //optional Bulk operations
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Reminder: Collection<E> Interface

```
public interface Collection<E> extends
                Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element); //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    //optional Bulk operations
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Collection Conventions

Variants

- No separate interface for each variant of each collection type
  - ***Immutable***, ***fixed-size***, ***append-only***, etc.

Collection Conventions

Variants

- No separate interface for each variant of each collection type
  - ***Immutable***, ***fixed-size***, ***append-only***, etc.
- **Optional** methods — a given implementation may select whether or not to support all operations

Collection Conventions

Variants

- No separate interface for each variant of each collection type
  - ***Immutable***, ***fixed-size***, ***append-only***, etc.
- **Optional** methods — a given implementation may select whether or not to support all operations
- Invoking an unsupported operation yields an **UnsupportedOperationException**
  - Implementations are responsible for documenting which optional operations they support

Lecture 7d: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

Lecture 7d: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

Packages

- Logical partition of code
  - A package is a namespace that organizes a set of related classes and interfaces
  - Like folders in your computer
  - Keeping the files in order
- Dividing the code into individual modules
- Adding permissions

Package Example

java.util

- The collection framework is actually a package (java.util)

Package Example

java.util

- The collection framework is actually a package (java.util)
- Its components (Interfaces, implementations, algorithms, etc.) all logically belong to the same package
  - They all share a very important semantic property – being closely related to data structures

Class Members Access Permissions

```
public class A{
        private int a;
        public int b;
        protected int c;
        int d;
        void foo(){
                int e;
        }
}
```

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
*default(=*package*)*	Y	Y	N	N
private	Y	N	N	N

Class Members Access Permissions

```
public class A{
        private int a;
        public int b;
        protected int c;
        int d;
        void foo(){
                int e;
        }
}
```

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
*default(=*package*)*	Y	Y	N	N
private	Y	N	N	N

Class Members Access Permissions

```
public class A{
        private int a;
        public int b;
        protected int c;
        int d;
        void foo(){
                int e;
        }
}
```

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default(=package)	Y	Y	N	N
private	Y	N	N	N

Class Members Access Permissions

```
package pack1;
public class A {
      int  packageInt;
}

package pack2;
import pack1.A;
class B {
      A a = new A();
      System.out.println(a.packageInt);
}
```

Modifier	Class	Package	Subclass	World
*default(=*package*)*	Y	Y	N	N

Class Members Access Permissions

```
package pack1;
public class A {
      int  packageInt;
}

package pack2;
import pack1.A;
class B {
      A a = new A();
      System.out.println(a.packageInt);
}
```

Modifier	Class	Package	Subclass	World
*default(=*package*)*	Y	Y	N	N

Class Members Access Permissions

```
package pack1;
public class A {
      int  packageInt;
}

package pack2;
import pack1.A;
class B {
      A a = new A();
      System.out.println(a.packageInt);
}
```

Modifier	Class	Package	Subclass	World
*default(=*package*)*	Y	Y	N	N

Error!

Class Members Access Permissions

```
package pack1;
public class A {
      int  packageInt;
}

package pack1;
class B {
      A a = new A();
      System.out.println(a.packageInt);
}
```

Modifier	Class	Package	Subclass	World
*default(=*package*)*	Y	Y	N	N

Class Members Access Permissions

```
package pack1;
public class A {
      int  packageInt;
}

package pack1;
class B {
      A a = new A();
      System.out.println(a.packageInt);
}
```

Modifier	Class	Package	Subclass	World
*default(=*package*)*	Y	Y	N	N

Class Members Access Permissions

```
package pack1;
public class A {
	protected int protectedInt;
}

package pack1;
class B {
	A a = new A();
	System.out.println(a.protectedInt);
}
```

Modifier	Class	Package	Subclass	World
protected	Y	Y	Y	N

Class Members Access Permissions

```
package pack1;
public class A {
	protected int protectedInt;
}

package pack1;
class B {
	A a = new A();
	System.out.println(a.protectedInt);
}
```

Modifier	Class	Package	Subclass	World
protected	Y	Y	Y	N

Package Classes

- Classes can be defined without any modifier

```
class MyClass { … }
```

Package Classes

- Classes can be defined without any modifier

```
class MyClass { … }
```

- Such classes are only accessible to other classes in the same package
  - Classes from other packages cannot use these classes

Package Classes

- Classes can be defined without any modifier

```
class MyClass { … }
```

- Such classes are only accessible to other classes in the same package
  - Classes from other packages cannot use these classes
- The **public** classes defined inside a package define its API

Exceptions and Packages

- Exceptions should be put in the same package as the classes that throw them
  - And not all in the same package
  - **IOException** resides in java.**io**
  - **EmptyStackException** resides in java.**util**
  - …

Exceptions and Packages

- Exceptions should be put in the same package as the classes that throw them
  - And not all in the same package
  - **IOException** resides in java.**io**
  - **EmptyStackException** resides in java.**util**
  - …
- This is because exceptions are **logically connected** to the classes that throw them
  - More than they are connected to one another

Lecture 7e: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

Lecture 7e: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes

Nested Classes

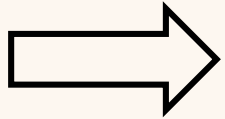
- A class within another class

```
class Class1 {
        ...
}
class Class2 {
        ...
}
```

Nested Classes

- A class within another class

```
class Class1 {
        ...
}
class Class2 {
        ...
}
```

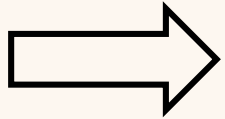


```
class Class1 {
        ...
        class Class2 {
                ...
        }
}
```

Nested Classes

- A class within another class

```
class Class1 {
        ...
}
class Class2 {
        ...
}
```



```
class Class1 {
        ...
        class Class2 {
                ...
        }
}
```

- Enclosing class ⇔ Wrapping class ⇔ Outer class

Nested Classes

Why?

- **Logical grouping of classes**
  - If a class is useful to only one other class, then it makes sense to embed it in that class and keep the two together

Nested Classes

Why?

- **Logical grouping of classes**
  - If a class is useful to only one other class, then it makes sense to embed it in that class and keep the two together
- **Increased encapsulation**
  - Consider two top-level classes (A, B), where B needs access to members of A that would otherwise be declared **private**
  - By hiding **class** B within **class** A, A's members can be declared **private** and B can **access them**
  - In addition, B itself can be hidden from the outside world

Nested Class

Why (2)?

- **More readable, maintainable code**
  - Nesting small classes within top-level classes places the code closer to where it is used

Nested Class

When?

- A nested class must be relatively small
  - A few small methods at most
  - Otherwise, this creates a readability problem

Nested Class

When?

- A nested class must be relatively small
  - A few small methods at most
  - Otherwise, this creates a readability problem
- It is generally not recommended to declare a nested class **public**
  - Although possible

Nested Class Privileges

- A nested class has access to any of its enclosing class members
  - Even if they are declared **private**
  - An enclosing class can also access a nested class's (**private**) members

Nested Class Privileges

- A nested class has access to any of its enclosing class members
  - Even if they are declared **private**
  - An enclosing class can also access a nested class's (**private**) members
- Nested classes can have any of the four access control modifiers
  - **private**, package, **protected** and **public**
  - Recall that top level classes can only be in **public** or **package** visibility

static Nested Class

- **static** nested classes are instantiated with **regardless of the existence** of instances of the **enclosing class**

static Nested Class

- **static** nested classes are instantiated with **regardless of the existence** of instances of the **enclosing class**
- Behaviorally, it is a **top-level class** that has been **nested** in another class for **packaging convenience**
  - A **static** nested class interacts with instances of its outer class (and other classes) just like any other top-level class
  - In addition, static nested classes may use **private** data (fields, methods, constructors) of objects of the outer class, and vice versa
    - The enclosing class can access the nested class data

static Nested Class

Creation Example

```
public class EnclosingClass {
    private int dataMember = 7;
    private static class NestedClass {
        private int nestedDataMember = 8;
    }
```

static Nested Class

Creation Example

```
public class EnclosingClass {
    private int dataMember = 7;
    private static class NestedClass {
        private int nestedDataMember = 8;
    }
```

static Nested Class

Creation Example

```
public class EnclosingClass {
    private int dataMember = 7;
    private static class NestedClass {
        private int nestedDataMember = 8;
    }
    public static public void main(String[] args) {
        // No need to create an instance of EnclosingClass
        NestedClass nested = new NestedClass();
        System.out.println(nested.nestedDataMember);
    }
}
```

static Nested Class

Privileges Example

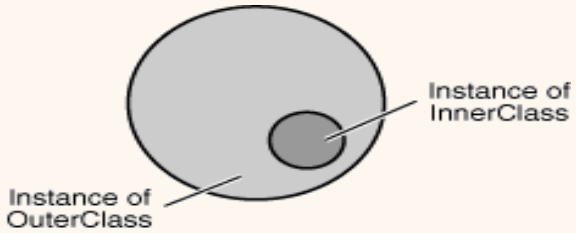
```
public class EnclosingClass {
    private int dataMember = 7;
    public void createAndIncrease()   {
            NestedClass in = new NestedClass();
            in.nestedDataMember++;              // a private field of the nested class
    }
    private static class NestedClass {
            private int  nestedDataMember = 8;
            private void nestedCreateAndIncrease() {
                EnclosingClass en = new EnclosingClass() ;
                en.dataMember++ ;               // a private field of the enclosing class
            }
     }
}
```

Inner Class

- Any **non-static** nested class
- Associated with *an instance **inst*** of its enclosing class
  - Has direct access to ***inst***'s members (even **private**)
  - Because an inner class is associated with an instance, it cannot define any **static** members
  - Instances of an inner class cannot be defined without an instance of the outer class

Member Class

- A member class is an inner class which is a member of the enclosing class
- To instantiate a member class:
  - Instantiate the outer class
  - Create the inner object within the outer object (e.g., within an instance [non-static] method, a constructor, etc.)



Member Class

Example

```
public class EnclosingClass {
    private class MemberClass {
        private int memberClassField = 8;
    }


}
```

Member Class

Example

```
public class EnclosingClass {
    private class MemberClass {
        private int memberClassField = 8;
    }

}
```

Member Class

Example

```
public class EnclosingClass {
    private class MemberClass {
        private int memberClassField = 8;
    }
    public void foo() {
        // a MemberClass object is created with reference
        // to a specific instance of OuterClass
        MemberClass innerObj = new MemberClass();
    }
}
```

Member Class

Privileges Example

```
public class EnclosingClass {
    private int dataMember = 7;
    public void createAndIncrease()   {
            MemberClass mem = new MemberClass();     // mem is associated with this
             mem.memberClassField++;                 // a private member of the member class
    }
    private class MemberClass {
            private int  memberClassField = 8;
            private void memberClassIncrease() {
                dataMember++ ;      // a private field of the enclosing class object associated with this inner obj
            }
     }
}
```

Exception as a Nested Class?

- In general, this is considered bad practice
- It’s better to implement exception classes in a separate file
  - Exception classes should be part of the package API
  - Implementing exceptions as nested classes would require **public** access (since external classes should know it and catch it)

Lecture 7f: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes
- Closures (Optional)

Lecture 7f: Overview

- Intro to Exceptions
- Exception Types
- Why Exceptions
- Packages
- Nested Classes
- Closures (Optional)

Local Classes

- A **local class** is an inner class which is declared locally within **a method**

Local Classes

- A **local class** is an inner class which is declared locally within **a method**
- Like a local variable, a local class is accessible only within the scope defined by its enclosing method

Local Classes

- A **local class** is an inner class which is declared locally within **a method**
- Like a local variable, a local class is accessible only within the scope defined by its enclosing method
- When do we use local classes?
  - If a member class is used only within a single method of its enclosing class

Local Class Example

```
public class StringLinkedList {
    private Node head;
    …
    public Iterator<String> elements() {          // This method creates and returns an Iterator object
        class ListIterator implements Iterator<String> {          // Definition of the local class
            Node current;
            ListIterator() { … } // inner class Constructor
            public boolean hasNext() { … }
            public String next () {… }
        } // end of local class ListIterator
        return new ListIterator();
    }
}
```

Local Class Example

```
public class StringLinkedList {
    private Node head;
    …
    public Iterator<String> elements() {          // This method creates and returns an Iterator object
        class ListIterator implements Iterator<String> {          // Definition of the local class
            Node current;
            ListIterator() { … } // inner class Constructor
            public boolean hasNext() { … }
            public String next () {… }
        } // end of local class ListIterator
        return new ListIterator();
    }
}
```

Local Class Example

```
public class StringLinkedList {
    private Node head;
    …
    public Iterator<String> elements() {          // This method creates and returns an Iterator object
        class ListIterator implements Iterator<String> {          // Definition of the local class
            Node current;
            ListIterator() { … } // inner class Constructor
            public boolean hasNext() { … }
            public String next () {… }
        } // end of local class ListIterator
        return new ListIterator();
    }
}
```

Local Class Example

```
public class StringLinkedList {
    private Node head;
    …
    public Iterator<String> elements() {          // This method creates and returns an Iterator object
        class ListIterator implements Iterator<String>{          // Definition of the local class
            Node current;
            ListIterator() { … } // inner class Constructor
            public boolean hasNext() { … }
            public String next () {… }
        } // end of local class ListIterator
        return new ListIterator();
    }
}
```

Privileges of Local Class

- As inner classes, local classes are associated with an enclosing instance, and can access any field or method (including **private**) of the wrapping class

Privileges of Local Class

- As inner classes, local classes are associated with an enclosing instance, and can access any field or method (including **private**) of the wrapping class
- In addition, local classes can access any local variables that are in the scope of the local method definition and are declared **final**
  - They cannot access non-**final** variables

Local Class Example

```
public class LinkedList {
    private Node head;
    public Iterator<String> elements() {
        final int LOCAL_START = 0; int localIndex = start;          // elements()'s local variables
        class ListIterator implements Iterator<String> {  //  a local class
            int index, start; Node iteratorHead;
            ListIterator() {                                        //  A method of  the local class
                start = LOCAL_START;                                // final local variable – legal code
                index = localIndex;                                 // Local variable – compilation error
                iteratorHead = head;                                // Accessing private data member – legal
            }
        } // end of local class ListIterator
        return new ListIterator();
    }
```

Local Class Example

```
public class LinkedList {
    private Node head;
    public Iterator<String> elements() {
        final int LOCAL_START = 0; int localIndex = start;        // elements()'s local variables
        class ListIterator implements Iterator<String> {  //  a local class
            int index, start; Node iteratorHead;
            ListIterator() {                                      //  A method of  the local class
                start = LOCAL_START;                              // final local variable – legal code
                index = localIndex;                               // Local variable – compilation error
                iteratorHead = head;                              // Accessing private data member – legal
            }
        } // end of local class ListIterator
        return new ListIterator();
    }
```

Local Class Example

```
public class LinkedList {
    private Node head;
    public Iterator<String> elements() {
        final int LOCAL_START = 0; int localIndex = start;          // elements()'s local variables
        class ListIterator implements Iterator<String> {  //  a local class
            int index, start; Node iteratorHead;
            ListIterator() {                                        //  A method of  the local class
                start = LOCAL_START;                                // final local variable – legal code
                index = localIndex;                                 // Local variable – compilation error
                iteratorHead = head;                                // Accessing private data member – legal
            }
        } // end of local class ListIterator
        return new ListIterator();
    }
```

Local Class Example

```
public class LinkedList {
    private Node head;
    public Iterator<String> elements() {
        final int LOCAL_START = 0 int localIndex = start;        // elements()'s local variables
        class ListIterator implements Iterator<String> {  //  a local class
            int index, start; Node iteratorHead;
            ListIterator() {                                  //  A method of  the local class
                start = LOCAL_START;                          // final local variable – legal code
                index = localIndex;                           // Local variable – compilation error
                iteratorHead = head;                          // Accessing private data member – legal
            }
        } // end of local class ListIterator
        return new ListIterator();
    }
```

Local Class Example

```
public class LinkedList {
    private Node head;
    public Iterator<String> elements() {
        final int LOCAL_START = 0; int localIndex = start;      // elements()'s local variables
        class ListIterator implements Iterator<String> {  //  a local class
            int index, start; Node iteratorHead;
            ListIterator() {                                   //  A method of  the local class
                start = LOCAL_START;                           // final local variable – legal code
                index = localIndex;                            // Local variable – compilation error
                iteratorHead = head;                           // Accessing private data member – legal
            }
        } // end of local class ListIterator
        return new ListIterator();
    }
```

Local Class Example

```
public class LinkedList {
    private Node head;
    public Iterator<String> elements() {
        final int LOCAL_START = 0; int localIndex = start;          // elements()'s local variables
        class ListIterator implements Iterator<String> {  //  a local class
            int index, start; Node iteratorHead;
            ListIterator() {                                        //  A method of  the local class
                start = LOCAL_START;                                // final local variable – legal code
                index = localIndex;                                 // Local variable – compilation error
                iteratorHead = head;                                // Accessing private data member – legal
            }
        } // end of local class ListIterator
        return new ListIterator();
    }
```

Restrictions on Local Class

- A local class is visible only within the block that defines it; it can **never** be used outside that block
  - We can use objects of the local class from outside that block, but only if they're up-casted to an external type

Restrictions on Local Class

- A local class is visible only within the block that defines it; it can **never** be used outside that block
  - We can use objects of the local class from outside that block, but only if they're up-casted to an external type
- A local class **cannot** be declared **public**, **protected**, **private**, or **static**

Restrictions on Local Class

- A local class is visible only within the block that defines it; it can **never** be used outside that block
  - We can use objects of the local class from outside that block, but only if they're up-casted to an external type
- A local class **cannot** be declared **public**, **protected**, **private**, or **static**
- As inner classes, local classes cannot contain **static** members

Restrictions on Local Class

- A local class is visible only within the block that defines it; it can **never** be used outside that block
  - We can use objects of the local class from outside that block, but only if they're up-casted to an external type
- A local class **cannot** be declared **public**, **protected**, **private**, or **static**
- As inner classes, local classes cannot contain **static** members
- Interfaces cannot be defined *local*

Anonymous Class

- An **anonymous class** is a local class without a name

Anonymous Class

- An **anonymous class** is a local class without a name
- An anonymous class is defined and instantiated in a single succinct expression using the **new** operator
  - Can be defined within any expression, such as inside a method call

Anonymous Class

- An **anonymous class** is a local class without a name
- An anonymous class is defined and instantiated in a single succinct expression using the **new** operator
  - Can be defined within any expression, such as inside a method call
- When a local class is used only once, consider using an anonymous class

Anonymous Class Example

- The ***File.list****(FilenameFilter filter)* method lists the files in a directory
- Before returning the list, it passes the name of each file to a *FilenameFilter* object you must supply
  - This *FilenameFilter* object accepts or rejects each file

```
interface FilenameFilter {
    boolean accept(File dir, String s);
}
```

Anonymous Class Example

- Since the body of the class is quite short, it is easily defined as an anonymous class

```
String[] filelist = dir.list(
        new FilenameFilter() {  // Creating an instance while
                                // implementing the class
        public boolean accept(File dir, String s) {
            return s.endsWith(".java");
        }
    } // end of class declaration
); // end of the statement of calling dir.list()
```

Anonymous Class Example

- Since the body of the class is quite short, it is easily defined as an anonymous class

```
String[] filelist = dir.list(
        new FilenameFilter() {  // Creating an instance while
                                // implementing the class
          public boolean accept(File dir, String s) {
              return s.endsWith(".java");
          }
       } // end of class declaration
); // end of the statement of calling dir.list()
```

Anonymous Class Example

- Since the body of the class is quite short, it is easily defined as an anonymous class

```
String[] filelist = dir.list(
        new FilenameFilter() {  // Creating an instance while
                                // implementing the class
        public boolean accept(File dir, String s) {
            return s.endsWith(".java");
        }
    } // end of class declaration
); // end of the statement of calling dir.list()
```

Note that the statement ends with a semi-colon

Closures

- A **"*closure*"** is a block of code (typically a function) that interacts with variables defined outside the block in a unique manner

Closures

- A **"*closure*"** is a block of code (typically a function) that interacts with variables defined outside the block in a unique manner
- Closures are typically implemented by nested functions (a function inside a function)

Closures

- A **"*closure*"** is a block of code (typically a function) that interacts with variables defined outside the block in a unique manner
- Closures are typically implemented by nested functions (a function inside a function)
- A closure function can access its non-local variables even when invoked outside of its declaration scope
  - Such functions may continue to live after the outer function terminates

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)             # prints 1
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)             # prints 1
closure1(7)
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)             # prints 1
closure1(7)             # prints 8
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)             # prints 1
closure1(7)             # prints 8
closure2(1)
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)             # prints 1
closure1(7)             # prints 8
closure2(1)             # prints 1
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)             # prints 1
closure1(7)             # prints 8
closure2(1)             # prints 1
closure1(1)
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)             # prints 1
closure1(7)             # prints 8
closure2(1)             # prints 1
closure1(1)             # prints 9
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)             # prints 1
closure1(7)             # prints 8
closure2(1)             # prints 1
closure1(1)             # prints 9
closure2(1)
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Closures Python Example

```
def counter():
    # A local variable
    x = 0

    # A nested function
    def increment(y):
        #    Access    non-local
        # variable
        nonlocal x
        x += y
        print(x)

    return increment
```

```
closure1 = counter()    # closures are put
closure2 = counter()    # into local variables

closure1(1)             # prints 1
closure1(7)             # prints 8
closure2(1)             # prints 1
closure1(1)             # prints 9
closure2(1)             # prints 2
```

http://en.wikipedia.org/wiki/Closure_(computer_science)

Local Classes as Closures

- On one hand, they are defined within functions and can access outer variables

Local Classes as Closures

- On one hand, they are defined within functions and can access outer variables
- On the other hand, while typical closures can modify non-local variables, in Java a local class has no access to **non-final** local variables of the enclosing method

Local Classes as Closures

- On one hand, they are defined within functions and can access outer variables
- On the other hand, while typical closures can modify non-local variables, in Java a local class has no access to **non-final** local variables of the enclosing method
- Java 8 introduced lambda expressions, which support closures

So far…

- Exceptions
  - Error handling
  - Checked vs. Unchecked
- Why
  - Separating error handling code from the rest of the code
  - Error propagating up the call stack
  - Grouping together and differentiating error types

So far…

- Packages
  - Helpful in organizing code
  - Can be used to restrict code access
- Nested class
  - Static classes
  - Inner classes
  - Further reading:
    - http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

Next Week

- Modularity
- More Design Patterns