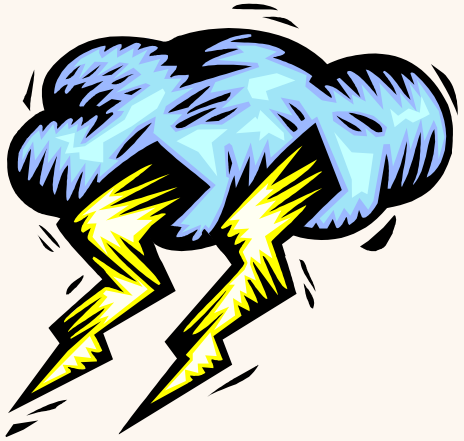
Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

Lecture 7

Exceptions



Nested classes

Errors

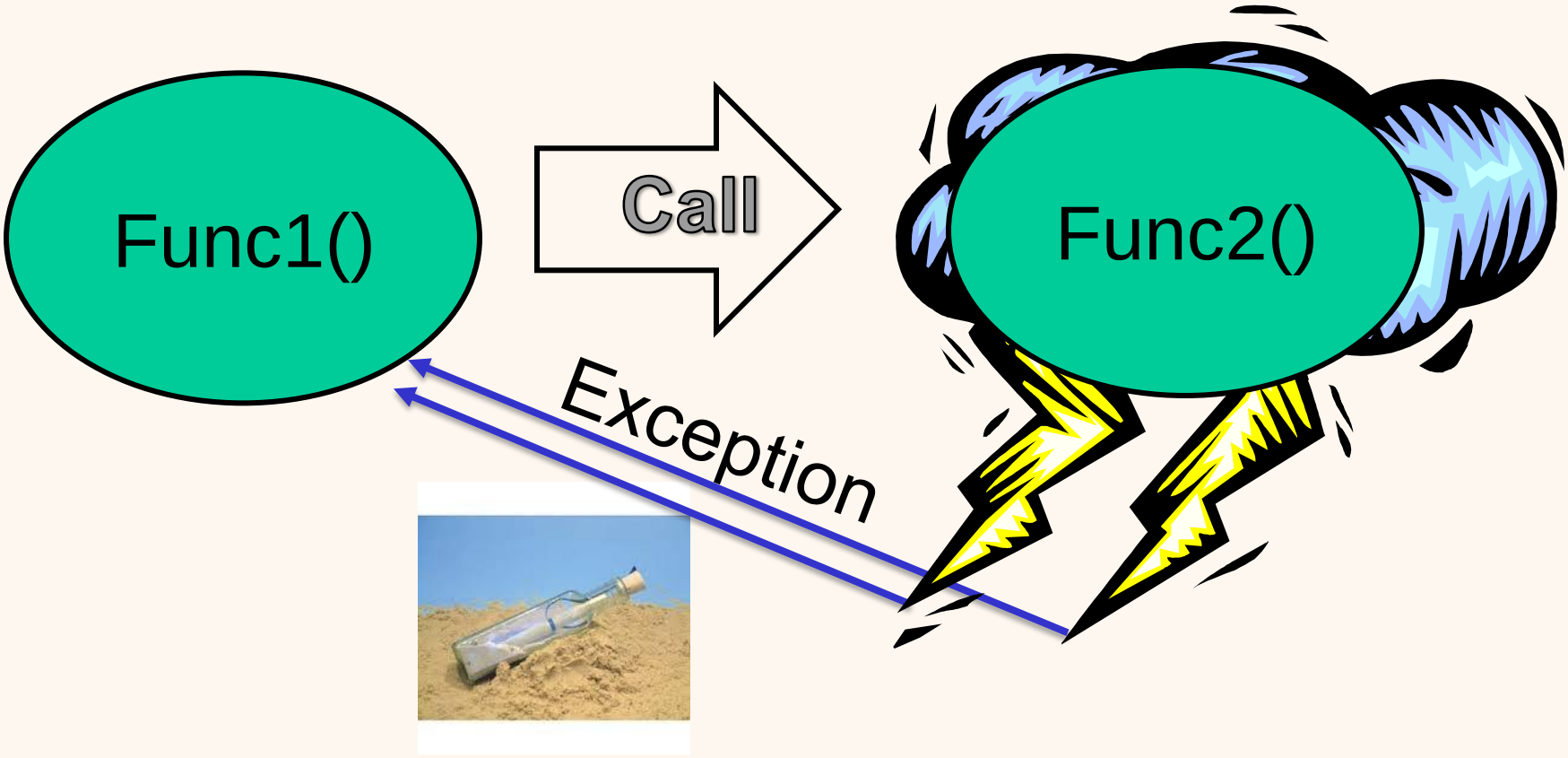
- Compilation errors
  - Detected by the compiler
- Runtime errors
  - **Not** detected by the compiler
  - Require error handling
  - Result of:
    - Bugs
    - Bad input
    - …

Runtime Error Handling

- Runtime error handling is a major issue in programming
- 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

Coffee Machine

Exceptions



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 what is the best way to handle this error
  - If it cannot handle it, it will send another **message** to its calling method
- In java, exceptions are **Objects**

Called Method Side: Throwing Exceptions

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

List of potential errors

Method halts, message is passed back to calling method

Specifying Exceptions

- **public void** foo() **throws** **Exc1, Exc2** {…}

- Part of the method declaration
  - Should be documented in the API
  - Use the tag @throws in javadoc

Calling Method Side: Handling Exceptions

```
public void foo() {
   try{
        //get index from user
        int element = list.get(0);
        //...
   } catch(ListException e){
        // Do something
   } catch(OtherException e){
        // Do something else
   }
   // Rest of method
}
```

Handling Exceptions

- A method that calls another method that **throws** an exception must either:
  - **catch** that exception (if it **knows** how to handle the error)
    - No cows found
  - **throw** this exception to its caller (if it **doesn't know** how to handle the error)
    - A missile
    - This is done by using the **throws** keyword (and not using **try**/**catch** blocks

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

Some Old Memories…

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\Research\Code>java exception1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
        at exception1.printFunc2(exception1.java:7)
        at exception1.printFunc1(exception1.java:10)
        at exception1.main(exception1.java:13)

c:\Research\Code>_
```

Exception “Not handled”

Hierarchy:

Exceptions are objects

SYSTEM ERRORS

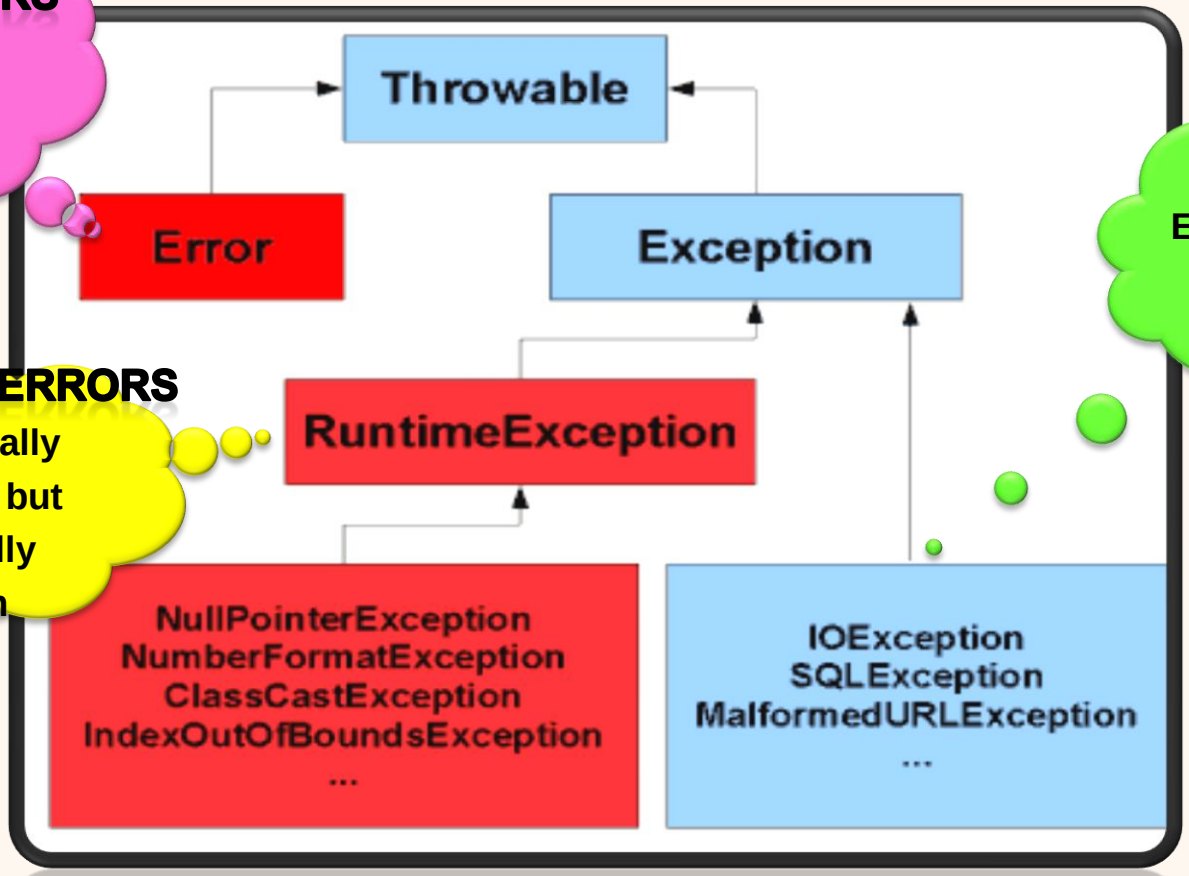
Shouldn't expect to catch and recover from

USER ERRORS

Expect to occur in the normal course of duty

PROGRAMMER ERRORS

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



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

Throwable Types

- **Checked**
  - Extend the **Exception** class
  - **Are** checked by the compiler
  - Usually result from *user* errors (wrong file name, bad URL, …)
    - IOException
  - Usually specific to that program
  - Checked exceptions must appear in the *throws* statement
    - 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
  - Could occur in many different programs and scenarios
  - **Are not** checked by the compiler
    - No need for the *throws* statement
    - For better documentation, specify non-trivial exceptions

Unchecked Exception Example

```
public void foo() {
   String tmp = null;
   System.out.println(tmp);
}

public void foo2() {
   foo();
}
```

- A NullPointerException is thrown, but not caught
- No compilation error (**unchecked** exception)
- foo2() can catch it, but is not required

Why Exceptions?

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

Grouping Error Types

```
class ListException extends Exception{…}

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

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

Grouping Error Types 2

```
public void foo() throws ListException() {
   try{
        //get index from user
        int element = list.get(0);
        //...
   } catch(ListException e){
        // Same error handling for all ListExceptions
   }
   // Rest of method
}
```

Separating Error Handling Code

- Consider the following operations on some data

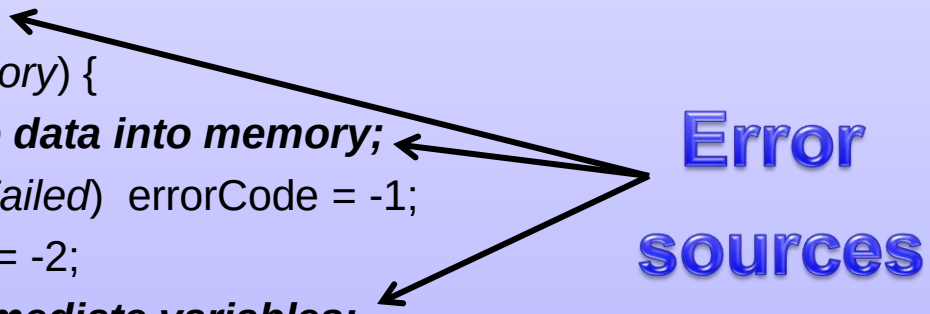
```
readData {
    ask for data size;
    allocate required memory;
    read the data from user into memory;
    cleanup unneeded variables;
}
```

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

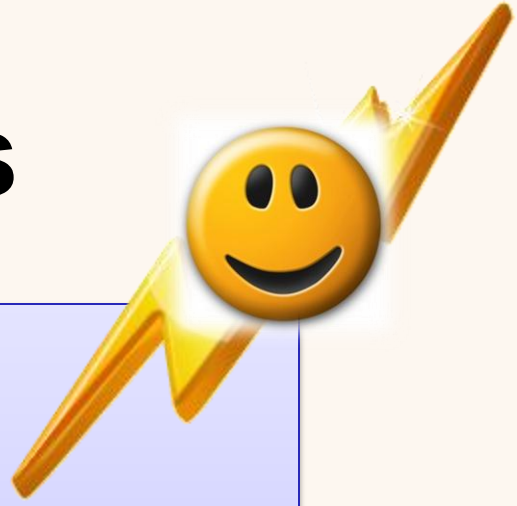
Without Exceptions

```
public int readData {
    int errorCode = 0;
    ask the data size;
    if (dataSizeIsOk) {
        allocate memory;
        if (gotEnoughMemory) {
            read the data into memory;
            if (readFailed)  errorCode = -1;
        } else  errorCode = -2;
        cleanup the intermediate variables;
        if (dataSizeIsOk && errorCode == 0)   errorCode = -3;
        else   …..;
    } else   errorCode = -4;

    return errorCode;
}
```



With Exceptions



```
void readData {
        try {
                ask for data size;
                allocate required memory;
                read the data from user into memory;
                cleanup unneeded variables;
        }

        catch (DataSizeException e) { doSomething; }
        catch (OutOfMemException e) { doSomething; }
        catch (ReadException e) { doSomething; }
        catch (CleanupException e) { doSomething; }
}
```

So far…

- Exceptions
  - Error handling
  - Checked vs. Unchecked

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

Nested Classes

- A class within another class

```
class OuterClass {
    ...
    class NestedClass {
      ...
    }
}
```

- Enclosing class ⇔ Wrapping class ⇔ Outer class

Nested Class

Why?

- **Logical grouping of classes**
  - If a class is useful to only one other class, then it is logical 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
- **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

- It is generally not recommended to declare a nested class public
  - Although optional

Static Nested Class

- Static nested classes are instantiated with **no dependence 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 the instance members of its outer class (and other classes) just like any other top-level class
  - In addition, a static nested class may use **private** members of instances of the outer class

Static Nested Class

Creation Example

```
public class EnclosingClass {
    private static class NestedClass {
        …
    }

    public static public void main(String[] args) {
        // No need to create an instance of EnclosingClass
        NestedClass in = new NestedClass();
    }
}
```

Static Nested Class

Privileges Example

```
public class EnclosingClass {
        private int dataMember = 7;
        public void createAndIncrease()   {
                NestedClass in = new NestedClass();
                in.innerDataMember++;   // a private member of the inner class
        }

        private static class NestedClass {
                private int  innerDataMember = 8;
                private void innerCreateAndIncrease() {
                        EnclosingClass en = new EnclosingClass() ;
                        en.dataMember++ ;              // a private member of the
                }                                      // enclosing class
        }
}
```

Inner Class

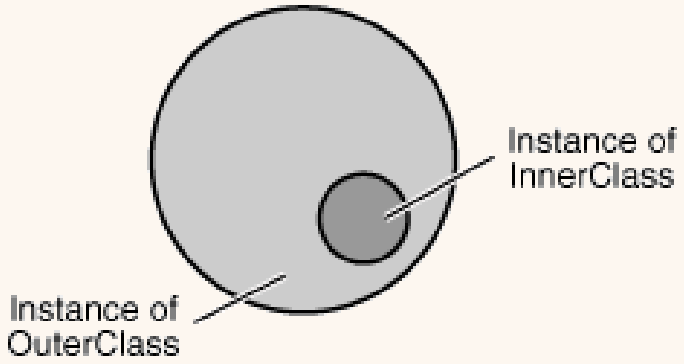
- Any **non-static** nested class

- Associated with *an instance* of its enclosing class
  - Has direct access to *that* object's methods and fields (even **private** fields)
  - Because an inner class is associated with an instance, it cannot define any **static** members
  - Objects that are instances of an inner class exist *within* an instance of the outer class

Member Class

- A member class is an inner class which is a member of the enclosing class

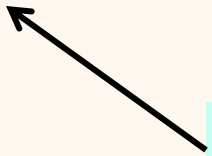
- To instantiate a member class:
  - Instantiate the outer class
  - Create the inner object within the outer object



Member Class
Example

```
public class OuterClass {
    private class InnerClass {
        …
    }

    public void foo() {
            // an InnerClass object is created with reference
            // to a specific instance of OuterClass (this)
            InnerClass innerObj = this.new InnerClass();
    }
}
```



Notice this syntax

Member Class Privileges

- A member class has unlimited access to its enclosing class members
  - Even if they are declared **private**
  - The enclosing class can also access the nested class's (**private**) members

- Like any other class member, member classes can have any access control modifiers
  - **private**, **protected** and **public**
  - Recall that top level classes **cannot** be declared **private** or **protected**

Member Class
Privileges Example

```
public class EnclosingClass {
        private int dataMember = 7;
        public void createAndIncrease()   {
                NestedClass in = this.new NestedClass(); // in is associated with this
                in.innerDataMember++;   // a private member of the inner class
        }

        private class NestedClass {
                private int  innerDataMember = 8;
                private void innerCreateAndIncrease() {
                        dataMember++ ;                  // a private member of the
                }                                       // enclosing class
        }
}
```

Static Nested Class

Usage Example

- LinkedList node
  - A node in a linked list is inherently related to the list class
    - It is also a small class (generally containing 2 methods, data() and next())
  - However, a node is not connected to a specific list
    - A node can be a member of more than one list
  - This is a good example of a static nested class

Inner Class
Usage Example

- List iterator
  - A list iterator is also inherently related to the list class
    - It is also a small class (generally containing 2 methods, next() and hasNext())
  - However, an iterator is connected to a specific list
    - An iterator is undefined without a given list
  - This is a good example of an inner member class

Exception as a Nested Class?

- Bad practice
- It's better to implement exception classes in a separate file
  - Implementing exceptions as nested classes would require a public access (since external classes should know it and catch it)
  - In addition, exception classes should be part of the API package

So far…

- Nested class
  - Static classes
  - Inner classes
    - member class, local class

- Further reading:
  - http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html