Design Patterns

Coding practices

Dr. Chad Williams
Central Connecticut State University

Documentation

- Many modern languages have tools/frameworks such as Javadoc for automated creation of **usable** code documentation
- Document purpose of classes and relations
- Document expectations of methods

```
/**
 * Javadoc comment used on classes or functions
 */

@param parameterName description
@return description of what is returned
@throws SomeException description of when
@author Name of author of class
```

Naming conventions

- Follow the naming convention of the language being used **they vary**

Examples following for Java expectations

- Class name should always begin with capital letter
- Object, methods and attribute names should always begin with lower case letter
- Multi-word names should capitalize first letter of each word (aka camelCase)

```
public class Point {
    Point myPoint = new Point();
}
```

Constants

Constants specified as final class fields:

```
public class Card{
  public final static int SUIT_SPADES=0;
  public final static int SUIT_HEARTS=1;
  public final static int
  SUIT_DIAMONDS=2;
  public final static int SUIT_CLUBS=3;
}
```

- Naming convention, all CAPS with underscore between words

What is a package

- Java packages consist of one or more classes
- Logical grouping of classes that are highly related, thus usually only a small number of classes per package
- Large projects sometimes consist of thousands of classes, packages allow you organize code
- Each Java class belongs to exactly one package

Design guideline

- When deciding which classes should be in the same package use this criteria
  - Are the classes closely related
  - Classes that change together should belong to the same package
  - Classes that are **not** reused together should **not** belong to the same package

Package specification

- First thing declared in source file of the form:

```
package edu.ccsu.cs407;
```

- If package is unspecified class belongs to unnamed package, not good style for keeping code organized
- Naming convention
  - all lower case
  - Widest to narrowest group from left to right

Using packages

- To use a class within the same package as the class you are in you simply name that class
- To use a class in another package you have to **import** it using its *fully qualified name*. There are two ways to do this
  - `import packagename.ClassName;`
  - `import packagename.*;`
- `java.lang` package imported implicitly by all java programs since it contains fundemental classes used in nearly all programs

Class modifiers

Basic

- `public` – accessible any class
- `protected` – accessible from subclasses and within package
- *none* - (none specified thus package) accessible from within package only
- `private` – accessible within class only

More advanced

- `abstract`
  - Some implementation
  - Contains abstract methods
  - Cannot be instantiated
  - Subclasses must implement abstract methods
- `final` – class may not be extended

Method/attribute

- public, protected, *none*, private
- `static` – Shared by all instances of the class, static methods can only access class static attributes
- `final`
  - method cannot be overriden
  - attribute constant value cannot be changed

Method only modifiers

- `abstract` – method must be implemented by subclasses
- `synchronized` – method atomic in multi-threaded environment
- `native` – method implemented in C/C++

Field only

- `volatile` – may be modified by non-synchronized methods in multi-threaded environment
- `transient` – field not part of persistent state of instance

Static fields

- By default when a field is declared it is an instance field
  - It only exist within the context of a single instance
  - Changes to it only reflected in that instance
- Static fields or class fields are specified by the
- `static` keyword
  - Field can be accessed without an instance of the class
  - Changes made by any instance are seen in all other instances

Exceptions

- Unexpected conditions in programs
- Exception handling mechanism to help recover or exit gracefully in the event of an unexpected condition or failure
- Critical to making large scale systems robust. Without fail something will go wrong and without exceptions very difficult to handle all possible scenarios in code.

Why special mechanism

1) Location where error occurs may not be the place you want to handle the exception – disrupts normal flow
2) Adding exception handling within normal logic makes the code unnecessarily complex
3) Ad hoc methods of error handling are often platform specific and non-portable

Sources of Exceptions

- Anytime normal flow of execution is interrupted. Usually originate from the following:
  - Run time environment – exception occurs due to illegal operation: Access operations or attribute on null pointer, specifying an index of an array that is out of bounds → RuntimeException
  - When an unexpected condition occurs in the code an exception is explicitly thrown with the `throw` statement

Meaning of exceptions

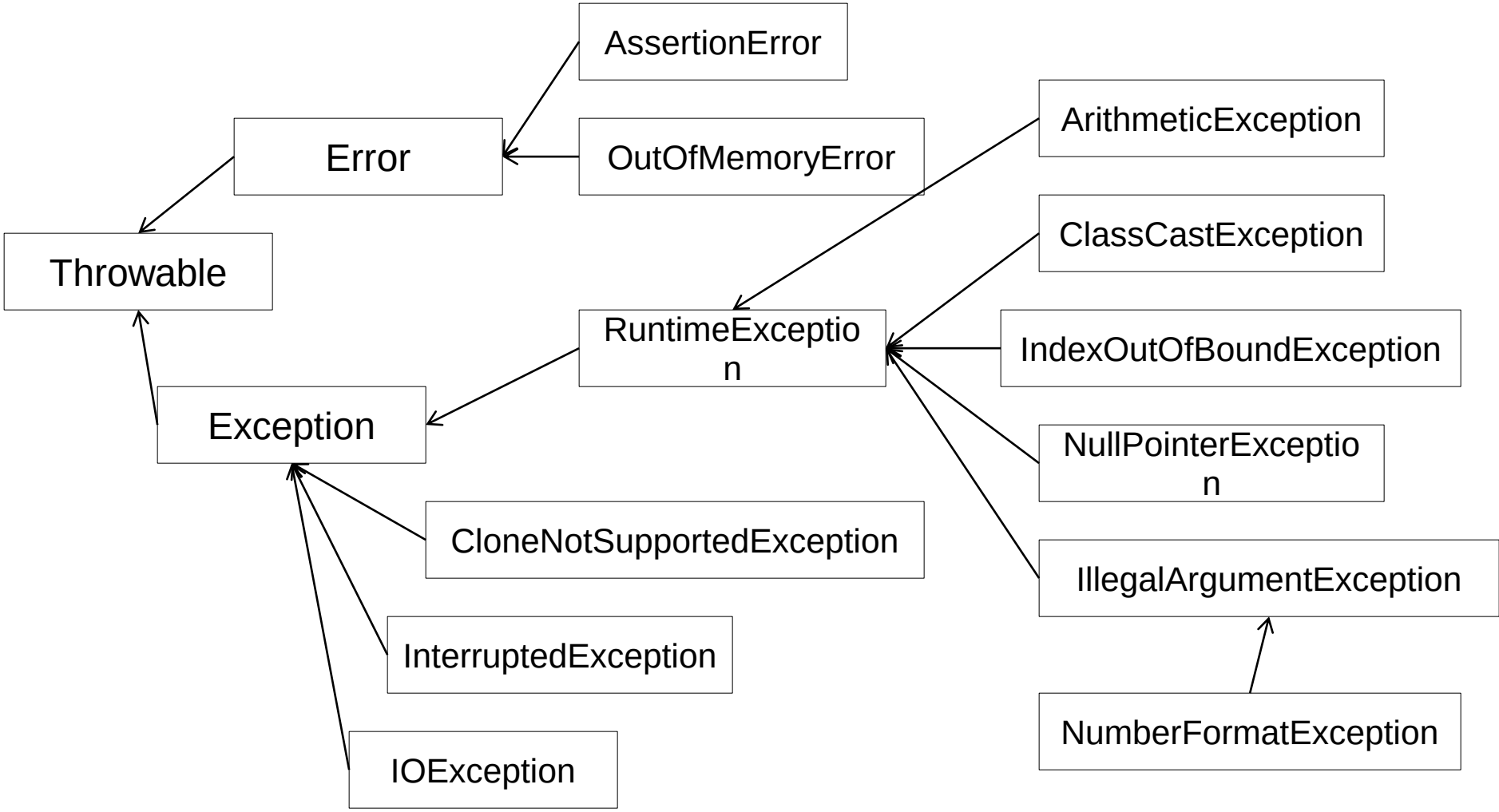
- Runtime errors generally represent a logical error that should be eliminated as part of debugging
- Exceptions meant to allow recovery not replace debugging
- In addition to the exceptions defined in `java.lang`, packages can declare their own exceptions such as `java.io.FileNotFoundException`
- You can also define your own exceptions

Hierarchy of Exceptions

- Throwable is parent class of all exceptions, a class must extend Throwable to be handled by exception handling

Category	Description
Error	Serious and fatal problems in a program, thrown by JVM and typically not handled by programs
Exception	Can be thrown by any program. All user-defined exception should extend Exception
RuntimeException	RuntimeExceptions represent an illegal operation and should be thrown by JVM

Exception hierarchy



Checked vs. unchecked

- Errors and run-time exceptions are called **unchecked exceptions**
  - Thrown by JVM in general these should all be able to be eliminated with thorough unit test. Handling usually handled at high level recovery
- All other exceptions are **checked exceptions**
  - Any function that does not explicitly handle a check exception must declare that it throws that exception

Declaring thrown exceptions

- For **checked exceptions** if the exception isn't handled it must declare that it throws that exception using the throws clause:
- `[Method modifiers] returnType methodName([parameters]) [throws Exception1, Exception2,...]`

```
if (passedAccount.funds < 0){
      throw new MyStateException();
}
```

Exception handling

- If method does not declare that it throws an exception that may be thrown within its method it must handle the exception itself

```
try{
    throw new MyException();
}catch(MyException e){
    //handle the exception
}
```

- Note a method may handle some exceptions but not all in which case it would need to declare the ones it didn't handle

Exception example

```
public void someMethod() throws
    InterruptedException, MyException
{
  // some statements
}


public void myMethod() throws MyException
{
  try{
    someMethod();
  }catch(InterruptedException e){
    // some action
  }
}
```

Handling exceptions

General ways of handling exceptions

1) Recover from the exception and resume execution

2) Throw another exception and pass the responsibility to another exception handler

3) Terminate the program gracefully when unable to recover

Best practices

- Generally better idea to specify which exception you want to catch rather than the general `Exception` class
- Likewise generally don't want to catch an exception and ignore it `}catch(Exception e){}`
- For a real application you are likely going to want to handle different exceptions in different ways depending on how severe they are.

Handling exception cont.

- With try block, each type of exception can be handled in a different manner

```
try{
   // statements
}catch(IOException e1){
   // handling
}catch(DBException e2){
   // handling
}catch(MyException e3){
   throws new MyOtherException("msg",e3);
}finally{
   // finish up which is optional
}
```

- Exceptions checked ***sequentially*** (important for subclasses)
- If the exception matches **any** of the catch blocks the `finally` block will be executed afterward before proceeding ahead with that directed by the `catch` block (ie. Rethrown, or continue execution).
- A typical use for `finally` is to close the database connection since you would want to do that regardless of which error

Debugging exceptions

- At runtime when an exception occurs the JVM will propagate the exception up the stack until it reaches the class that handles the exception
- For debugging it is very useful to print the stack trace of an exception which will tell you the entire path of the exception from where it occurred all the way back to the main method with line numbers associated with each call:

```
}catch(MyException e){
    e.printStackTrace();
}
```

Sequence diagrams – exception flow

- Players:
  - App
  - AccountHandler
    - createAccount throws ConnectionFailException
  - DBWriter
    - insertAccount throws DupAcctException, ConnectionFailException

Group work

Create 3 classes A, B, C (you can assume the Exception classes are already defined). Class A has a method *operationA* that takes two arguments (doubles) *a* and *b* and returns a double. The function should calculate the (square root of *a)*/*b*. If *a* is negative it should throw NegAException, if *b* is zero it should throw BZeroException. Class B should have a method *operationB* that calls the method on Class A and catches just the NegAException and prints a message indicating *a* can't be negative. Class C should call Class B's method and if BZeroException is thrown it should output stack debug information.

(Math function is “sqrt”)

Group work

An instant messaging application where the user can create a person to person connection or connect to a chat, where there is a chat moderator

- State diagram
- Class diagram
- Sequence diagram
  - *Consider exception that could occur along your sequence*

Well behaved classes

All well behaved classes

- Object equality
  - equals()
    - Instance equality vs Object equality
  - hashCode()
- String representation

Supporting object equality

- There are a number of principles that must hold for object equality

- **Reflexivity –** for any object x, `x.equals(x)` must be true
- **Symmetry –** for any objects x and y, `x.equals(y)` is true iff `y.equals(x)`
- **Transitivity –** for any objects x, y and z, if both `x.equals(y)` and `y.equals(z)` then `x.equals(z)`
- **Consistency –** for any objects x and y, `x.equals(y)` should consistently return true or false
- **Nonnullity –** for any object x, `x.equals(null)` should return false

Typical equality methods

```
public boolean equals(Object other){

  if (other == null){ return false;}

  if (this == other){

    return true;  //same instance
  }else if(other instanceof C){

    C otherObj = (C) other;

    // compare each field, if there are

    // differences return false else return true

  }

  return false;

}
```

Comparison of fields

- Primitive types

```
if (p != otherObj.p) return false;
```

- Reference types

```
if (r==null){
  return (otherObj.r ==null);
}else{
  return r.equals(OtherObj.r);
}
```

- Note that some fields may be temporary or not important in which case they do not need to be part of the comparison

Hash code of objects

- `hashCode()` method is used by hash tables as their hashing function
- A hash code has the following properties:
  - if `x.equals(y), x.hashCode()` must equal `y.hashCode()`
  - However `x.hashCode()` equaling `y.hashCode()` does not mean x and y are equal

hashCode implementation

- A common way to compute hash codes is to take the sum of all the hash codes that are significant fields on the object

```
public int hashCode(){
   int hash = 0;
   hash += primitiveType;
   hash += refType.hashCode();
}
```

- For something like a linked list where there are many elements that make its significant fields, a common approach is to take the hash of the first *x* fields. This will ensure equality/hash code relationship is maintained while also reducing time to build hash.

Expectations going forward

- All code turned in for this class must follow these best practices in order to receive full credit
  - **Javadoc** for all classes and public methods (include param, return, throws tags)
  - **Naming conventions** – packages, classes, methods, attributes, constants
  - **Packages – use them default "no package" not acceptable**, purposeful naming, module organization
  - **Visibilities** – all visibilities method/attribute should be chosen with encapsulation/modularity in mind (always better to err on side of too restrictive then widen only as needed)
  - **Exceptions** – throw catch specific checked exceptions, if intentionally ignore exception add comment as to this intent and why
  - **Well behaved classes**
    - Meaningful toString always
    - Meaningful equals() / hashCode() method if possibility of more than one instance