CS 417
Design Patterns

Testing work
Being well behaved
Singleton

Dr. Chad Williams
Central Connecticut State University

When catch an exception, when you wouldn't want to

- Exception indicates a problem in execution - bad input, invalid unexpected condition
- Would catch an exception when you want that function to handle the error
- Would not catch an exception i.e. let the exception flow through if the exception should be handled in a higher level calling function
- Exception do not necessarily have anything to do with letting the user know what happened, they could be handled quietly in the code, recover and resume execution

Group work

- Blackbox test cases for Stack implementation
  - Methods on Stack
    - size()
    - push(Object obj)
    - pop(Object obj) throws NoMoreElementsException
    - toString()
- For each be specific about setup/initialization and steps to test each scenario

Group work

- **White box testing -** Test cases for following implementation – be specific

```
public boolean doSomething(int a, int b, boolean c)
throws BadException{
  if (((a<b)&&(c==true))||((b>a)&&(c==false)){
   return otherCall(c)  //returns true or false based on c
  } else if (b>a){
     throw new BadException();
  }else{
   return true;
  }
}
```

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.

Design patterns

- Pattern describes problem that occurs **over and over** again and the **solution** to that problem
- Solution can be applied to all the situations even though surrounding implementation changes
- Not language specific

Design patterns cont.

- Design patterns classified into 3 categories
  - **Creational patterns** – deal with process of creating objects
  - **Structural patterns** – deal primarily with the static composition and structure of classes and objects
  - **Behavioral patterns** – deal primarily with dynamic interaction among classes and objects

Description of design pattern

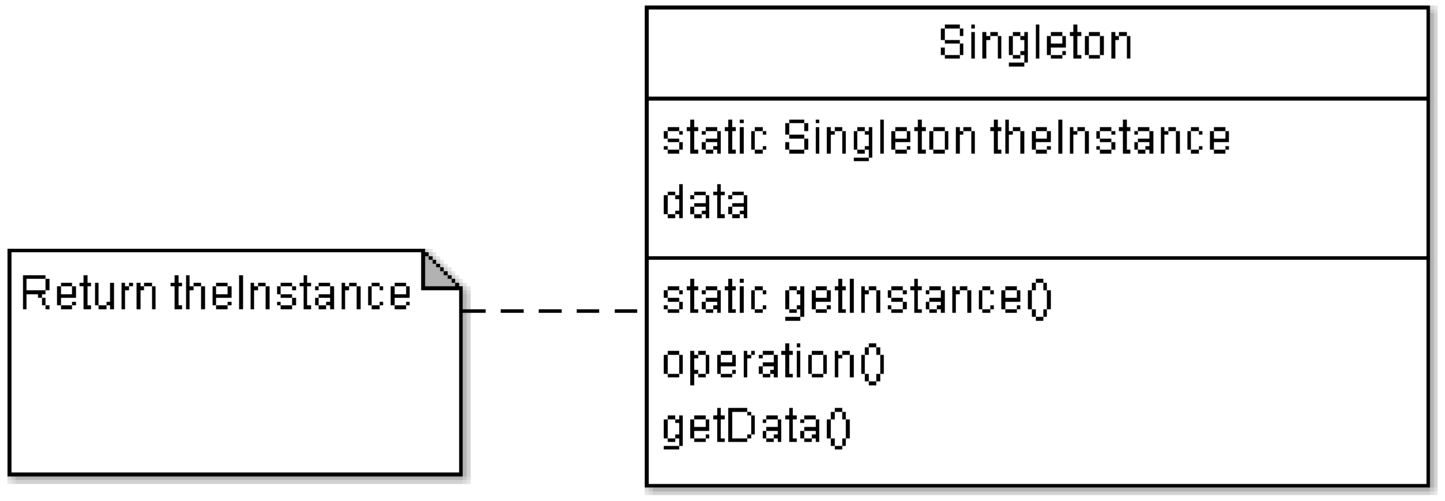
Consist of the following:

- **Pattern name** – essence of the pattern
- **Category** – Creational, structural, behavioral
- **Intent** – Short description of design issue the problem addresses
- **Also known as** – other names for the pattern
- **Applicability** – Situations when pattern can be applied
- **Structure** – class or object diagram that depicts participants and relationships
- **Participants** – List of classes and/or objects participating in the problem

Design pattern: Singleton

- **Category:** Creational design pattern
- **Intent:** Ensure class has only one instance and provide global point of access to it
- **Applicability:** Use when there must be exactly one instance of a class, accessible to clients from a well-known access point
- **Participants:** Only one participant

Singleton structure



Examples of use

- Logger – have a single point of access to the file that handles writing to a log file. Avoids multiple open file handles inconsistent state (flush not immediate)
- Dictionary or code lookup – only one mapping needed, also can be expensive to create multiple instances

Implementation

```
public class Singleton{
  public static Singleton getInstance(){
    if (theInstance == null){
      theInstance = new Singleton();
    }
    return theInstance;
  }
  private Singleton(){
    // initialize singleton fields
  }
  private static Singleton theInstance = null;
```

Drawbacks

- Unit testing more difficult due to global state of application
- Potential problems with parallel execution due to all interacting with same object simultaneously

Group work

- Identify a concrete example of when you might want to use a singleton
- Justify why you think so
- Create UML for your class including all attributes and methods and visibilities
- Create sequence diagram of “Class1” requesting instance of singleton followed by “Class2” requesting instance of singleton