Design Patterns

Chain of Responsibility wrap up
Well behaved classes

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

- A bank has a very sophisticated system to give their customers protection against bouncing checks by taking advantage of the number of accounts they hold as well as credit options.
  - Customers have 1 checking account, then optionally a savings account, and optionally 1 or more credit cards with different interest rates
  - When the CheckProcessor processs a customer's check it first checks if there are sufficient funds in the checking account if so stops, if not if the person has a savings account that is checked, otherwise try each of the customers many credit cards (lowest interest first)
- Create UML
- Create sequence diagrams for situations: checking acct sufficient; savings acct present but must go to credit; no savings but 3 credit cards none of which are sufficient

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.