# Design Patterns

## 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

# 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