# CS151

Software Design

Java Generics

Generic Bags

# Software Design Goals

writing good code

- Robustness
  - software capable of error handling and recovery
  - programs should never crash
    - ending abruptly is not crashing
- Adaptability
  - software able to evolve over time and changing conditions (without huge rewrites)
- **Reusability**
  - same code is usable as component of different systems in various applications
  - The story of Mel — https://www.cs.utah.edu/~elb/folklore/mel.html

# OOP Design Principles

- Modularity
  - programs should be composed of "modules" each of which do their own thing
    - each module is separately testable
  - Large programs are built by assembling modules
  - Objects (Classes) are modules
- Abstraction
  - Get to the core — non-removable essence of a thing
  - Most pencils are yellow, but yellowness does not required. They do have a color
- **Encapsulation**
  - Nothing outside a class should know about how the class works.
    - For instance, does the Object class have any instance variables. (Of what type?)
  - Allows programmer to totally change internals without external effect
    - What is external? By one definition, all vars should be "private"

The correct level of modularity and abstraction is debatable; in the end these are a programming choice

## OOP Design

- Responsibilities/Independence: divide the work into different classes, each with a different responsibility and are as independent as possible
- Behaviors: define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.
  - Recall: Java Interfaces define ONLY behaviors

## Software design

- Good variable names
- Comments
- In Java
  - Avoid statics
  - Minimize main
  - Use inheritance and class design

# Think before coding

- Point of UML (and one of the points of Java interfaces) is to get you to think about a problem before writing code
- Please do so
- While writing code,
  - get up and walk about
  - talk to a classmate about your thoughts
- Start early … please

# Java Interfaces

- No data fields
- No constructors
- No private methods
- No protected methods
- No bodies for methods

- Lots of instructions about how the IO behavior of methods
- I will tend to use Java interfaces rather than UML

```
/**
 * Interface definition for Bag
 * Adapted slightly from Carrano & Henry
 * @author GTowell
 * Created: July 2021
 */
public interface BagOfPets {
    /**
     * The number of pets in the bag
     * @return the number of pets in the bag
     */
    public int numberOfItems();

    /**
     * true if there is at least one pet in the ba
     * @return true if there is at least one pet i
     */
    public boolean isEmpty();

//etc
```

# BagOfPets Interface

Same things as UML, just in java syntax

Every method documented, a lot!!

```
public interface BagOfPets {
    /**
     * The number of pets in the bag
     * @return the number of pets in the bag
     */
    public int numberOfItems();
    public boolean isEmpty();
    public boolean add(Pet p);
    public Pet remove();
    public boolean remove(Pet p);
    public void clear();
    public int countOf(Pet p);
    public boolean contains(Pet p);
    public void display();
}
```

# Implementing BagOfPets

- java
  - `public X implements Y`
  - This says making a class that will provide bodies for EVERY method in interface Y
  - Possibly more methods
    - private or protected helpers for public
    - private instance variables

```
/**
 * An implementation of the BagOfPets interface
 *
 * Note that anything marked with @Override does not
 * need documentation as it should be documented
 * elsewhere. Unless implemtation is not per doc
 * @author gtowell
 * Created: July 2021
 *
 */
public class PetBag implements BagOfPets {


    @Override
    public int numberOfItems() {
```

# BagOfPets & PetBag

- Design Goals:
  - robustness
    - Good (probably)
  - adaptability
    - poor (only pets)
  - reusability
    - poor (only pets)
- Design principles
  - Modularity
    - OK
  - Abstraction
    - poor (only pets)
  - encapsulation
    - OK
- Conclusion: These kind of suck!

```
public class PetBag implements BagOfPets {
    /** The array holding the information in the bag */
    private Pet[] petArray;

    /**
     * The default constructor.
     * Creates a bag that can hold 100 pets.
     */
    public PetBag() {
        this(100);
    }
    /**
     * Constructor for pet bag
     * param sizeOfBag  is the size of the bag
    */
    public PetBag(int sizeOfBag) {
        petArray = new Pet[sizeOfBag];
    }
```

# IN CLASS

- Implement the following methods for ObjectBag
  - numberOfItems: int
    - the number of pets in the bag
  - empty: boolean
    - Does the bag have any items
  - clear: void
    - Remove all items from the bag

# Generify code

- Idea: write code without being tied to Pets
- Approach 0
  - Replace every mention of Pet with Object.
    - Since all class inherit from Object, can put anything into bag.
    - Redefinition works!
- Until Java v5 this was only solution
  - ability to put ANYTHING into Bag can cause problems at run time

```
public class Bag implements BagOfObjects {
    /** The array holding the information in the bag */
    private Object[] obArray;

    /**
     * The default constructor.
     * Creates a bag that can hold 100 things.
     */
    public ObjectBag() {
        this(100);
    }
    /**
     * Constructor for bag
     * param sizeOfBag  is the size of the bag
    */
    public ObjectBag(int sizeOfBag) {
        obArray = new Object[sizeOfBag];
    }
```

## Generics

- Idea: want Bag to store anything, BUT only one kind of anything at a time.
- Let the specific thing be “bound” at compile time
  - Avoid a lot of run-time problems
- Java: Generics
  - Same idea appears in lots of other languages, with slightly different syntax

## Generic Interface

- Note the **<S>**
- This indicates a "generic"
  - By Convention: Generic indicated by any single capital letter
- Then "S" is used in rest of interface where it was "Pet"

```
public interface BagOfStuff<S> {
   public int numberOfItems();
   public boolean isEmpty();
   public boolean add(S p);
   public S remove();
   public boolean remove(S p);
   public void clear();
   public int countOf(S p);
   public boolean contains(S p);
   public void display();
}
```

# Generic Class

- Two uses of <R>
- After that, again, replace all mentions of “Pet” with “R”
- One trick: making generic array.

```
public class StuffBag<R> implements BagOfStuff<R> {
    /** The array holding the information in the bag *
    private R[] stuffArray;

    /**
     * The default constructor.
     * Creates a bag that can hold 100 stuff.
     */
    public StuffBag() {
        this(100);
    }

    /**
     * Constructor for stuff bag
     * param sizeOfBag  is the size of the bag
    */
    @SuppressWarnings("unchecked")
    public StuffBag(int sizeOfBag) {
        stuffArray = (R[])new Object[sizeOfBag];
    }
```

# Generic Bag Shelter

- Variable declaration
  - says that this instance of StuffBag can only hold Pet
    - and descendents
    - auto cast
- Variable Creation
  - actually make an instance of StuffBag that holds only Pets
- Access
  - Get a Pet
    - The instance still knows what it is, but the code does not.
    - So to do something specific, need to check then cast.
      - Cannot be automatic
      - instanceof

```
public class Shelter {
    // the store for the animals in the shelter
    private StuffBag<Pet> animals;
    public Shelter() {
        animals = new StuffBag<Pet>(100);
    }
    public void addAnimal(Pet animal) {
        animals.add(animal);
    }
    public Pet adoptRoulette() {
        return animals.remove();
    }
    @Override
    public String toString() {
        return animals.toString();
    }
    public static void main(String[] args) {
        Shelter shelter = new Shelter();
        shelter.addAnimal(new Dog("dave", "toy"));
        shelter.addAnimal(new WorkingDog("Jane", "BorderCo
        shelter.addAnimal(new Cat("Calypso", "1", "Siberia
        Pet aa = shelter.adoptRoulette();
        if (aa instanceof Cat) {
            Cat c = (Cat) aa;
            System.out.println("I Got a Cat!!!!" + c + aa
        }
        System.out.println(aa);
        System.out.println(shelter);
    }
}
```

## Classes with multiple Generics

- You can have many
- You can have some generic and some not

```
public class KeyValue<U, V> {
    private final U key;
    private final V value;
    public KeyValue(U key, V value) {
        this.key = key;
        this.value = value;
    }
    public U getKey() {
        return key;
    }
    public V getValue() {
        return value;
    }
    @Override
    public String toString() {
        return "<" + key + ", " + value + ">";
    }

    public static void main(String[] args) {
        KeyValue<String, Integer> ksvi = new KeyValue<>("key",
        KeyValue<Double, StringBuffer> kdvsb = new KeyValue<>(
StringBuffer("Now is the time"));
        System.out.println(ksvi);
        System.out.println(kdvsb);
    }
}
```