CS151

Software Design

Java Generics

Generic Bags

Software Design Goals

- 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
- **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

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.

Software design: Already discussed

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

Class Definition

- Primary means for abstraction in OOP
- Class determines
  - the way state information is stored – via instance variables
  - a set of behaviors – via methods
- Classes encapsulate
  - `private` instance variables
  - `public` accessor methods (getters)

Java Specifics
Constructors

- Constructors are never inherited
- A class may invoke the constructor of the class it extends via a call to `super` with the appropriate parameters
  - e.g. `super()`
  - `super` must be in the first line of constructor
  - If no explicit call to `super`, then an implicit call to the zero-parameter `super` will be made
- A class make invoke other constructors of their own class using `this()`
  - `this` must be first
  - Cannot explicitly use both `super` and `this` **in single constructor**

BagOfPets & PetBag

- Design Goals:
  - robustness
    - Good
  - adaptability
    - poor
  - reusability
    - poor
- Design principles
  - Modularity
    - OK
  - Abstraction
    - poor
  - encapsulation
    - not great
- 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];
    }
```

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 ObjectBag 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 OO languages, with slightly different syntax
  -

Generic Interface

- Note the <S>
- This indicates a "generic"
  - 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

```
public class GBShelter {
    // the store for the animals in the shelter
    private StuffBag<Pet> animals;
    public GBShelter() {
        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) {
        GBShelter shelter = new GBShelter();
        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);
    }
}
```

In Class

- Use StuffBag to store KeyValue pairs
- Adapt stuffBag to only take one instance of a given object
  - that is, a set rather than a bag
  - use equals not ==
- Adapt KeyValue so that equals tests for same key rather than same object