CS206

More Inheritance

Generics
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](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.
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
- See FileOpen.java for example
try/catch — with resources

```java
public void readOneLineTC(String filename) {
    BufferedReader br;
    try {
        br = new BufferedReader(
            new FileReader(filename));
        br.readLine();
    } catch (FileNotFoundException fnf) {
        System.err.println(
            "No file " + e);
    } catch (IOException e) {
        System.err.println(
            "Reading " + e);
    }
    finally {
        if (br!=null) {
            try {
                br.close();
            } catch (IOException ioe) {
                System.err.println("Close" + ioe);
            }
        }
    }
}
```

```java
public void readOneLineTCR(String filename) {
    try (BufferedReader br = new BufferedReader(
        new FileReader(filename))) {
        br.readLine();
        // close unnecessary in this formulation
    } catch (FileNotFoundException e) {
        System.err.println("Open " + e);
    } catch (IOException e) {
        System.err.println("Reading " + e);
    }
}
```

Finally == code that WILL be executed

Close can throw an exception so it too must be caught

See FileOpen.java
Method Overriding

• Inherited methods from the superclass can be redefined/changed
  □ “signature” stays the same
    □ signature = name+type of all args
• The appropriate version to call is determined at run time
• Most common overrides
  • toString
  • equals
DogDriver

DogDriver.java
Parsing strings

• Split method of String
  
  `string.split(String regexp)`

• split a string into an array of Strings based on matching delimiter. Then go through the array appropriately

  `StringSplitter.java`
Generics

• A way to write classes and methods that can operate on a variety of data types without being locked into specific types at the time of definition

• Write definitions & implementations with “Generic” parameters

• The generics are instantiated (locked down) when objects are created
import java.util.Random;
/*--------------------------*/
* @author gTowell
* Created: August 28, 2019
* Modified: Jan 24, 2019
* Purpose:
* Generic Methods
*--------------------------*/
public class GenericMethod {
    public static void main(String[] args) {
        Integer[] jj = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // NOTE AUTOBOXING!!!
        new GenericMethod().randomize(jj);
        for (int j : jj)
            System.out.println(j);
        String[] ss = { "A", "B", "C", "D", "E", "F" };
        new GenericMethod().randomize(ss);
        for (String s : ss)
            System.out.println(s);
    }

    public <T> void randomize(T[] data) {
        Random r = new Random();
        for (int i = 0; i < data.length; i++) {
            int tgt = r.nextInt(data.length);
            swap(data, tgt, i);
        }
    }

    — generic swap method
    — use reflection to check class
}
import java.io.BufferedReader;
import java.io.StringReader;
/**
 * Simple generic class example
 * @author gtowell
 * @param <A>
 */
public class GenericClass<A> {
    /** A non-generic value */
    private double amount;
    /** A generic value */
    private A otherValue;
    /** Constructor. */
    * @param other the generic value
    * @param amt a double value.
    */
    public GenericClass(A other, double amt) {
        this.otherValue = other;
        this.amount = amt;
    }

    public static void main(String[] args) {
        GenericClass<String> gString = new GenericClass<String>("ASDF", 24.5);
        System.out.println(gString);
        GenericClass<Double> gDouble = new GenericClass<Double>(99.5, 44.5);
        System.out.println(gDouble);
        GenericClass<BufferedReader> gBR = new GenericClass<BufferedReader>(
                new BufferedReader(new StringReader("When in the course")), 99.8);
        System.out.println(gBR);
    }
}

write a toString function for this class
Generics Restrictions

• No instantiation with primitive types
  • Genre<Double> ok, but Genre<double> is not

• Can not declare static instance variables of a parameterized type

• Can not create arrays of parameterized types
  • but you can create an array of Object then cast
    • (T[])new Object[10]
Nested Class

- A class defined inside the definition of another class
- When defining a class that is strongly affiliated with another
  - help increase encapsulation and reduce undesired name conflicts.
- Nested classes are a valuable technique when implementing data structures
  - represent a small portion of a larger data structure
  - an auxiliary class that helps navigate a primary data structure
  - **ONLY** place that public instance variables are acceptable
    - They aren’t really public
Nested Class Example

ClassNester.java