CS151 Intro to Data Structures

Java Review, Inheritance, Generics

Announcements

- ~apoliak/handouts/cs151
  - Goldengate is having issues
  - Will post on course website until issues are fixed

- Midterm: Wednesday 25th (Wednesday after Fall break)
  - Post on piazza if you want this to be pushed back
  - Lets resolve this by end of this week

Announcements

- Piazza:
  - Asynchronous communication
- Gradescope:
  - Submit all assignments
  - Can request re-grade requests

- If not on either, come to my office right after class

Announcements

- HW00 due tomorrow (09/12) night
- Lab00 – overview of command line and vim
- Lab01 must be checked off by a TA by Wednesday night

Outline

- Files & Exceptions (Lab01)
- Object Oriented Programming
- Generics

File I/O

1. import packages

```
import java.io.*
import java.util.*
```

2. Create a new `Scanner` object linked to the file we want to read

```
Scanner input = new Scanner(new File(<filename>));
```

3. Use `hasNextLine()` and `nextLine()` methods to read line by line until done

```
while(input.hasNextLine()) {
  String line = input.nextLine();
  ...
}
```

4. Close

```
input.close;
```

Exceptions – way to deal with unexpected events during execution

- Unexpected events:
  - unavailable resource
  - unexpected input
  - logical error

- Exceptions are objects that can be *thrown* by code expecting to encounter it

- An exception may also be *caught* by code that will handle the problem

Catching Exceptions

- Exception handling
  `try-catch`

- An exception is caught by having control transfer to the matching `catch` block

```
try {
    guardedBody
} catch (exceptionType1 variable1) {
    remedyBody1
} catch (exceptionType2 variable2) {
    remedyBody2
} ...
    ...
```

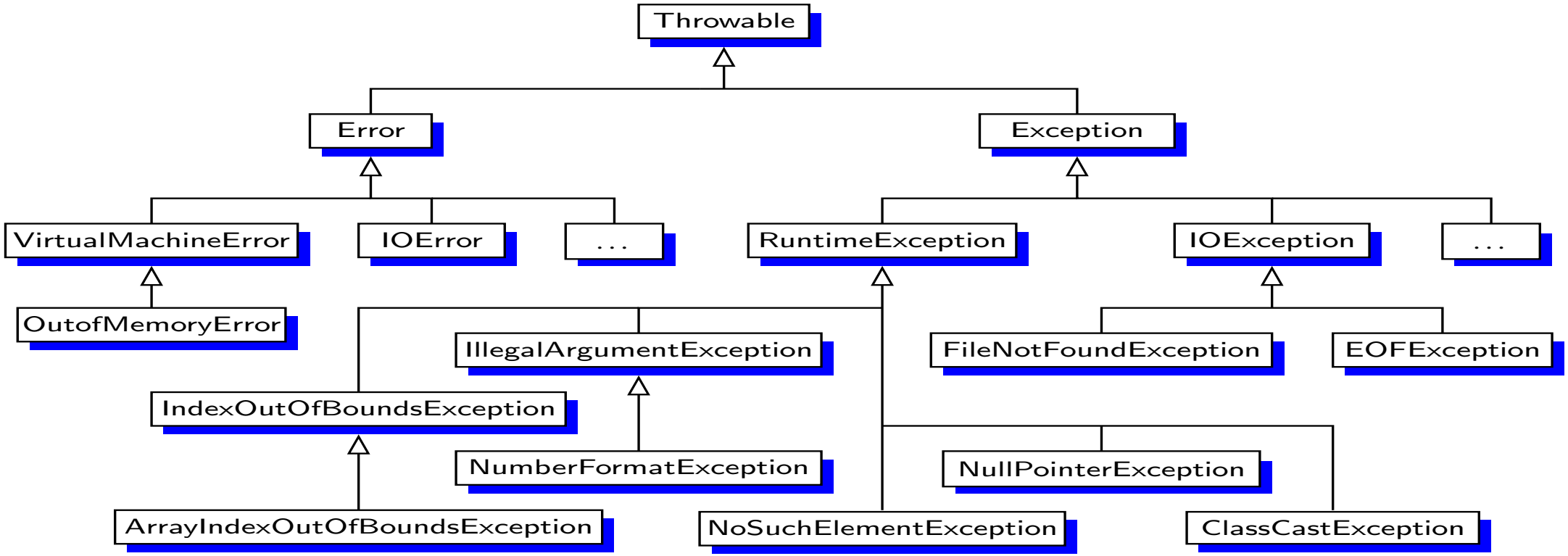
- If no exception occurs, all `catch` blocks are ignored

Throwing Exceptions

- An exception is thrown
  - implicitly by the JVM because of errors
  - explicitly thrown by code
- Exceptions are objects
  - throw an existing/predefined one
  - make a new one
- Method signature – `throws`

```
public static int parseInt(String s) throws
NumberFormatException
```

Java's Exception Hierachy



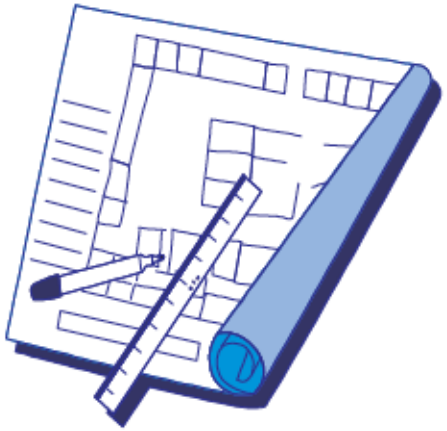
Software Design Goals

- Robustness
  - software capable of error handling and recovery
- 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

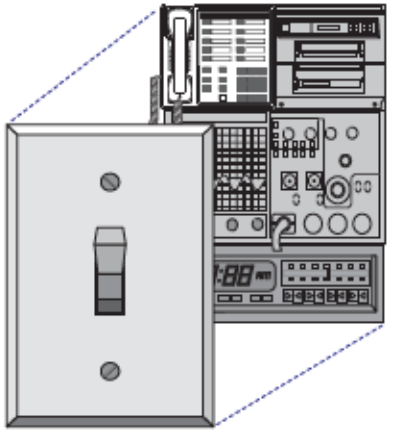
Object Oriented Programming Principles

- Modularity
- Abstraction
- Encapsulation

Modularity



Abstraction



Encapsulation

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.

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
- Class encapsulates
  - `private` instance variables
  - `public` accessor methods (getters)

Example

```
class Student {
  private String name;
  private int id;

  public Student(String name, int id) {
    this.name = name;
    this.id = id;
  }

  public String getName() {return name;}
  public int getId() {return id;}
}
```

Representing Objects

- What happens if we `System.out.println(obj)`?

```
Student s = new Student(”Ada Lee”, 1234);
System.out.println(s); //??
```

  - Prints location of the object in memory

- `toString()`
  - Special method in a class that provides a way to customize printing objects
  - returns a `String` representation of the instance object that is used by `System.out.println`
  - `public String toString()`

Student

```
class Student {
  private String name;
  private int id;
  // constructor and getters not shown

  public String toString() {
    return name+" "+id;
  }
}
```

Inheritance

- Allow a new class to be defined based on an existing class
  - Existing: base, super or parent class
  - New: subclass or child class

- Keyword `extends`

```
class CSStudent extends Student{ … }
```

- `CSStudent` inherits all `public` and `protected` instance variables and methods of `student`

Constructors

- Constructors are never inherited
- A subclass may invoke the superclass constructor via a call to `super` with the appropriate parameters
- If calling `super`, it must be in the first line of the subclass' constructor
- If no explicit call to `super`, then an implicit call to the zero-parameter `super()` will be made

CSStudent

```
class CSStudent extends Student{
  private boolean isMajor;
  public CSStudent(String name, int id, boolean isMajor){
    super(name, id);
    this.isMajor = isMajor;
  }
  public boolean getIsMajor() {return isMajor;}
}
CSStudent s1 = new CSStudent(”Adam Po", 1111, true);
CSStudent s2 = new CSStudent("Di Xu", 2222, false);
System.out.println(s1);
System.out.println(s2);
```

Output

```
Adam Po 1111
Di Xu 2222
```

Method Overriding

- Inherited methods from the superclass can be redefined/changed
  - signature stays the same
- The appropriate version to call is determined at run time
- `toString` is overridden, twice!
- All classes inherit from `Object`, which contains a `toString`

Method Overloading

- Overloading occurs when two methods have the same name but different parameters

- Determined at compile time

```
int a(int x) ;
int a(int x, int y);
int a(float y);
int a();
```

`super`

- `super` refers to the superclass object
- can also be used to reference methods defined in the superclass
  - usually because you overrode it
- `super.toString()`

`protected`

- access modifier
  - `public` – world
  - `private` – super class only
  - `protected` – super and subclasses
- subclass inherits all `public` and `protected` instance variable and methods
- What about `private` instance variables?

Type Hierarchy

- Every subclass object is an instance of its superclass
- A superclass object is NOT an instance of the subclass

```
class A {}
class B extends A {}
class C extends B {};
A a1 = new B();
B b1 = new A();
A a2 = new C();
B b2 = new C();
C c1 = new B();
C c2 = new A();
```

Type Hierarchy

- Every subclass object is an instance of its superclass
- A superclass object is NOT an instance of the subclass

```
class A {}
class B extends A {}
class C extends B {};
A a1 = new B();
B b1 = new A();
A a2 = new C();
B b2 = new C();
C c1 = new B();
C c2 = new A();
```

Homogeneous Type

- Array requires that the elements are of the same type

```
int[] nums = {1, 2, 3};
```

- A subclass object is an instance of its superclass

```
A[] abcs = new A[3];
abcs[0] = new A();
abcs[1] = new B();
abcs[2] = new C();
```

Object Casting

- Type conversion between super and subclasses – like the primitive types
- A superclass is a wider type
- A subclass is a narrower type

```
class A {}
class B extends A {}
class C extends B {};
B b1 = (B) new A();
C c1 = (C) new B();
C c2 = (C) new A();
A a1 = new B();
B b2 = (B) a1;
```

- Explicit super to sub cast is dangerous

Object Casting

- Type conversion between super and subclasses – like the primitive types
- A superclass is a wider type
- A subclass is a narrower type

```
class A {}
class B extends A {}
class C extends B {};
B b1 = (B) new A();
C c1 = (C) new B();
C c2 = (C) new A();
A a1 = new B();
B b2 = (B) a1;
```

- Explicit super to sub cast is dangerous

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.

OOP Design

- Instance variables keep track of the states of an object
  - the only place data is stored
- Methods assume all instance variables are always up-to-date
  - someone else will call the appropriate setter/updater
- Each method is responsible for one task and updating the related variables only

Source Code Organization

- Each project under its own subdirectory
  - directory name = project name
  - A1, A2, …
- One class per file - `public`
- name of the file matches class name
- `Driver.java`
- compiling just `Driver.java` usually compiles all

Generics

Write a class that supports an self-expanding array

```
public class ExpandingArray {
  private int[] array;
  public ExpandingArray(int size) {
    this.array = new int[size];
  }
  public void insert(int item) { … }
  public int getItem(int index) { … }
  public int indexOf(int item) { … }
}
ExpandingArray obj = new ExpandingArray(10);
```

Self-expanding Array class to support Strings

```
public class ExpandingArray {
  private String[] array;
  public ExpandingArray(int size) {
    this.array = new String[size];
  }
  public void insert(String item) { … }
  public String getItem(int index) { … }
  public int indexOf(String item) { … }
}
ExpandingArray obj = new ExpandingArray(10);
```

Generics

- A way to write classes or methods that can operate on a variety of data types without being locked into specific types at the time of definition
- Write definitions with type parameters
- The types are instantiated (locked down) when objects are created

Self-expanding Array as Generic Class

```
public class ExpandingArray<T> {
  private T[] array;
  public ExpandingArray(int size) {
    this.array = new int[size];
  }
  public void insert(T item) { … }
  public T getItem(int index) { … }
  public int indexOf(T item) { … }
}
ExpandingArray<String> obj1 = new ExpandingArray<String>(10);
ExpandingArray<Integer> obj1 = new ExpandingArray<Integer>(10);
```

Technically this is wrong, we'll see why in a few slides

Generic Class

```
public class Pair<A, B> {
  private A first; private B second;
  public Pair(A first, B second) {
    this.first = first; this.second = second;
  }
  public A getFirst() {return first;}
  public B getSecond() {return second;}
  public String toString() {//??}
}
Pair<String, Double> deposit =
new Pair<>("USD", 500.00);
```

Generic Method

```
public static <T> void reverse(T[] data){
  int low = 0; int high = data.length-1;
  // swap the ends towards the middle
  while (low < high) {
    T tmp = data[low];
    data[low] = data[high];
    data[high] = tmp;
    low++; high--;
  }
}
```

Generics Restrictions

- No instantiation with primitive types
- Can not declare static instance variables of a parameterized type
- Can not create arrays of parameterized types

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