CS206

Exceptions, Scope & Restaurants

Exceptions

- Try-catch or throws
  - Catch must tell user what / why
- If throws, somewhere **in the program** it must be caught
- Try-catch should be as "tight" as possible
- Programs should never die on an exception
  - It is not acceptable to simply surround main with try-catch

Exceptions
Bad = thrown but not caught

```
public class CatchErr1 {

    private void printArray(int[] data) throws
ArrayIndexOutOfBoundsException {
        for (int i = 0; i <= 10; i++) {
            System.out.println(data[i]);
        }
    }

    public static void main(String[] args) throws
ArrayIndexOutOfBoundsException {
        new CatchErr1().printArray(new int[5]);
        System.out.println("Done printing the array!");
    }
}
```

Exception
Bad = General + Loose + no message

```
public class CatchErr2 {
    /**
     * Poor exception handling
     * 1. The try catch block encloses whole method
     * 2. The catch block is empty.
     * 3. The exception caught is non-specific
     * @param args ignored
     */
    public static void main(String[] args) {
        int[] a = new int[10];
        try {
            for (int i = 0; i <= 10; i++) {
                System.out.println(a[i]);
            }
            System.out.println("Done printing the array!");
        } catch (Exception e) {
        }
    }
}
```

Exceptions
OK — but not good or great

```
public class CatchErr3
{
    public static void main(String[] args)
    {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        int prev=10;
        try {
            while (prev>=0) {
                System.out.print("Enter a number: ");
                String line = br.readLine();
                int data = Integer.parseInt(line);
                System.out.println(data + " / " + prev + " = " + (data /
prev));
                prev=data;
            }
        }
        catch (IOException ioe){
            System.err.println("Problem reading from keyboard" + ioe);
        }
        catch(NumberFormatException e) {
            System.out.println("That's not a number!");
        }}}
```

try should be “tighter”

Some exceptions not handled

Multiple try-catch

```
public class CatchErr4
{
    public static void main(String[] args
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int prev=10;
        while (prev>=0) {
            try {
                System.out.print("Enter a number: ");
                String line = br.readLine();
                try {
                    int data = Integer.parseInt(line);
                    System.out.println(data + " / " + prev + " = " + (data / prev));
                    prev=data;
                } catch(NumberFormatException e) {
                    System.err.println("That's not a number!");
                } catch (ArithmeticException ae) {
                    System.err.println("Division by Zero" + ae);
                }

            } catch (IOException ioe){
                System.err.println("Problem reading from keyboard" + ioe);
            }}}
```

Potential Problem Here

THIS IS NOT A GOOD WAY TO HANDLE DIVISION BY ZERO

Avoiding Exceptions

- Exceptions are very expensive!
  - So avoid if possible / practical
  - `RuntimeException` – always avoidable, but worth it?
  - For example:

```
try {
      int data = Integer.parseInt(line);
      System.out.println(data + " / " + prev + " = " + (data / prev));
      prev=data;
  } catch(NumberFormatException e) {
      System.err.println("That's not a number!");
  } catch (ArithmeticException ae) {
      System.err.println("Division by Zero" + ae);
  }
```

RuntimeException

Instead Write:

```
try {
    int data = Integer.parseInt(line);
    if (prev==0)
       System.out.println("Skipping to not divide by zero");
    else
       System.out.println(data + " / " + prev + " = " + (data / prev));
    prev=data;
} catch(NumberFormatException e) {
    System.err.println("That's not a number!");
```

RuntimeException (s)

Class RuntimeException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, CompletionException, ConcurrentModificationException, DataBindingException, DateTimeException, DOMException, EmptyStackException, EnumConstantNotPresentException, EventException, FileSystemAlreadyExistsException, FileSystemNotFoundException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, IllformedLocaleException, ImagingOpException, IncompleteAnnotationException, IndexOutOfBoundsException, JMRuntimeException, LSException, MalformedParameterizedTypeException, MalformedParametersException, MirroredTypesException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NoSuchMechanismException, NullPointerException, ProfileDataException, ProviderException, ProviderNotFoundException, RasterFormatException, RejectedExecutionException, SecurityException, SystemException, TypeConstraintException, TypeNotPresentException, UncheckedIOException, UndeclaredThrowableException, UnknownEntityException, UnmodifiableSetException, UnsupportedOperationException, WebServiceException, WrongMethodTypeException

Writing code with Exceptions

- Exceptions are Expensive
  - Sometimes it is correct to throw and exception in code you write, sometimes not
  - Usually it is debatable
    - which implementation of `first()` is better?

```
public class LinkedList<T> implements LinkedListInterface<T> {

  private Node<T> head = null;
  private Node<T> tail = null;
  private int size = 0;
  public int size() {return size;}
  public boolean isEmpty() {return size == 0;}

  public T first() throws NoSuchElementException {
    if (isEmpty()) {
          throw new NoSuchElementException("No Rabbits");
    }
    else {
        return head.element;
    }
  }
```

```
public class LinkedList<T> implements LinkedListInterface<T> {

  private Node<T> head = null;
  private Node<T> tail = null;
  private int size = 0;
  public int size() {return size;}
  public boolean isEmpty() {return size == 0;}

  public T first() {
    if (isEmpty()) {
          return null;
    }
    else {
        return head.element;
    }
  }
```

Scope

- Scope is used to define the "lifetime" of a variable.
- "Global" means variable always available from anywhere
  - public static
- "Local" means variable only available in a specific place.
  - { } delimit scope
- Each scope is aware of its variables
- Variables defined within a scope die at the end of the scope.
- Java does not allow var name re-use in enclosing scopes
  - Except public static
  - Except instance variables

Scope example

```
public class ScopeTest {
    private int var = 1; // instance variable -- horribly named

    public void scopes(int vv) {
        System.out.println(var);
        {
            int var = vv+2;
            System.out.println(var);
        }
        System.out.println(var);
    }
    public static void main(String args[]) {
        ScopeTest s = new ScopeTest();
        s.scopes(2);
    }
}
```

GT Restaurant

- GT offers 3 food types
    - drink :orangina, coffee, …
    - main course: burger, hot dog, …
        - Gluten Free: burger on lettuce
    - salad: spinach, cobb, …
- At the start of each day, GT decides what will be offered that day and how much is available, price & cost
- During day:
    - Order one item
        - How many
    - if run out, remove from list
    - if order would use more than available, reject/suggest getting less
- End of day print out:
    - leftover food
    - Cost, Revenue, Profit.

Considerations to implement GT

- How to store available menu items
  - GT wants to use only 1 data structure
- How to represent available menu items
- How/what to update on each order

GTRestaurant start

```
public class GTRestaurant
{
    public void addItem(Object o) {
    }
    public boolean doOrder(int id, int count) {
       return true;
    }
    public void endOfDay(){
        System.out.println("No cost, no revenue, no food");
    }
}
```

GT Restaurant Usage

```
public class Main {
    public static void main(String[] args)
    {
    GTRestaurant gtr = new GTRestaurant();
    // Add items at start of day
    gtr.addItem(new Drink(1, “sip orangina", 1.50, 0.45, 20, 20));
    gtr.addItem(new Drink(2, “small coffee", 2.99, 0.20, 4, 250));
    gtr.addItem(new MainC(101, "hamburger", 3.50, 2.20, 30));
    gtr.addItem(new MainC(102, "hot dog", 2.45, 1.05, 40));
    gtr.addItem(new MainCGF(103, "burger on lett", 5.99, 0.40, 2000));
    gtr.addItem(new Salad(201, "spinach", 2.50, 1.99, “Vinegar”, 20));
    gtr.addItem(new Salad(202, "cobb", 4.99, 1.99, “lemon juice”, 200));
    gtr.doOrder(1, 2);
    gtr.doOrder(101, 5);
    gtr.endOfDay();
    }}
```

MenuItems
Class Hierarchy