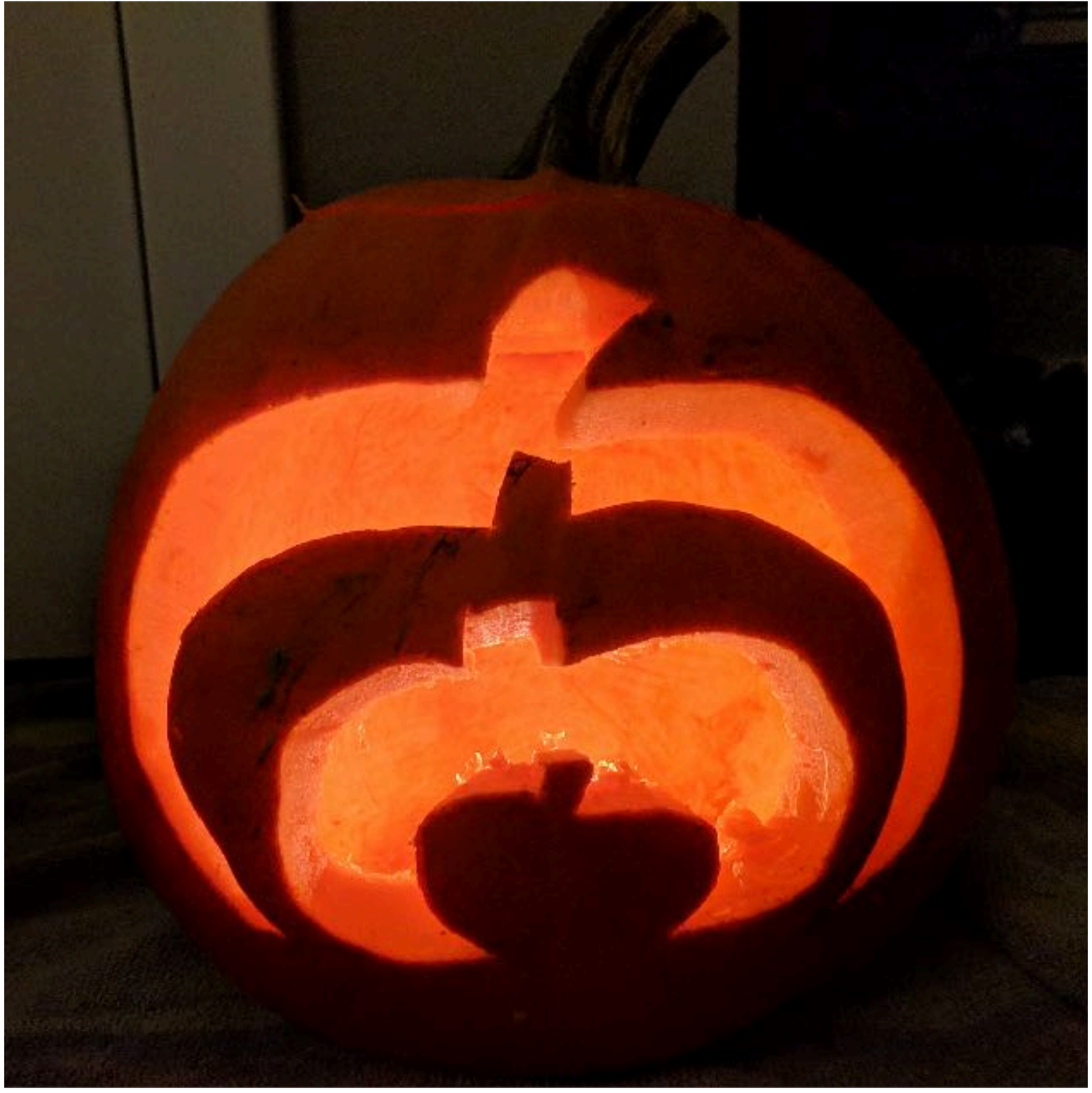

Lecture 16
Some Java Stuff:
Inheritance, Generics, Exceptions

Announcements

- Lab 2 is graded, grades are on Canvas.
- A1 is taking a while. Aiming for early next week.
- A2 slip days: the code I pulled at the deadline is what will be graded unless you told me you took a slip day.

To submit your work late, you must push your changes via git (as usual) *then* send me an email stating that you have submitted the assignment late. The timestamp of the *email*, which must be sent after your final changes are pushed to git, will be used as the submission time.

Goals

- Understand inheritance in Java.
- Know how to use and implement simple **Generic** classes.
- Know how to catch and throw exceptions.
- Be ready for the midterm exam.

Inheritance in Java

- A class can **extend** another class and **inherit** all of its public and protected methods.
- If a class does not extend any other class, it extends `Object` **by default.**
- Object has some methods:
  - https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html
  - `equals(), toString(), hashCode(), …`

public class A

```
public class A {

    // pretty exciting, eh?

}
```

A Very Brief Intro to Generics

Before Generics

```
interface Collection {
    /** Return true iff the collection contains ob */
    boolean contains(Object ob);
   /** Add ob to the collection; return true iff
      * the collection is changed. */
    boolean add(Object ob);
    /** Remove ob from the collection; return true iff
      * the collection is changed. */
    boolean remove(Object ob);
  ...
}
```

Can contain anything that extends Object (any class at all!)

- But **not primitive types**: int, double, float, boolean, …

Before Generics

```
interface Collection {
    /** Return true iff the collection contains ob */
    boolean contains(Object ob);
  /** Add ob to the collection; return true iff
     * the collection is changed. */
    boolean add(Object ob);
    /** Remove ob from the collection; return true iff
      * the collection is changed. */
    boolean remove(Object ob);
  ...
}
```

Can contain anything that extends Object (any class at all)

- But **not primitive types**: int, double, float, boolean, …

The Problem

```
Collection c = ...
c.add(“Hello”)
c.add(“World”);
...
for (Object ob : c) {
  String s = (String) ob;
  // do things with s
}
```

Notice: Arrays don’t have this problem!

```
String[] a = ...
a[0]= (“Hello”)
a[1]= (“World”);
...
for (String s : a) {
  System.out.println(s);
}
```

The Solution: Generics

```
Object[] oa= ...                    // array of Objects
String[] sa=  ...                   // array of Strings
ArrayList<Object> oA= ... // ArrayList of Objects
ArrayList<String> oA= ... // ArrayList of Strings
```

Now the Collection interface is implemented like this:

```
interface Collection<T> {
  /** Return true iff the collection contains x */
  boolean contains(T x);

  /** Add x to the collection; return true iff
    * the collection is changed. */
  boolean add(T x);

  /** Remove x from the collection; return true iff
    * the collection is changed. */
  boolean remove(T x);
  ...
}
```

The Solution: Generics

The Collection interface is now implemented like this:

```
interface Collection<T> {
  /** Return true iff the collection contains x */
  boolean contains(T x);

  /** Add x to the collection; return true iff
    * the collection is changed. */
  boolean add(T x);

  /** Remove x from the collection; return true iff
    * the collection is changed. */
  boolean remove(T x);
  ...
}
```

Key idea: I don’t need to know what T is to implement these!

The Solution: Generics

Key idea: I don't need to know what T is to implement these!

```
Collection<String> c= ...
c.add("Hello")   /* Okay */
c.add(1979);     /* Illegal: compile error! */
```

Generally speaking,

`Collection<String>`

behaves like the parameterized type

`Collection<T>`

where all occurrences of `T` have been replaced by `String`.

The Solution: Generics

The bummer: T must extend Object - no primitive types.
Can’t do:

```
Collection<int> c = ...
```

Have to use:

```
Collection<Integer>
```

Java often seamlessly converts `int` to `Integer` and back.

```
Integer x = 5; // works
int x = new Integer(5); // works
```

“Autoboxing/unboxing”

LinkedList<T>

- We often use an *inner class* to store Node objects of trees, graphs, lists, etc.

- It's defined inside the LinkedList class, and only used within the class.

The Comparable Interface

```
class Orange implements Comparable;
class Apple implements Comparable;
Orange o = new Orange();
Apple a = new Apple();
a.compareTo(o);
```

- We can compare apples to oranges!

```
interface Comparable<T> {
   int compareTo(T o);
}
```

The Comparable Interface

```
interface Comparable {
  public int compareTo(Object o)
}
```

- We can compare apples to oranges!

```
interface Comparable<T> {
  int compareTo(T o);
}
```

The Comparable Interface

```
interface Comparable<T> {
   int compareTo(T o);
}
```

```
class Orange implements Comparable<Orange>;
class Apple implements Comparable<Apple>;
Orange o = new Orange();
Apple a = new Apple();
a.compareTo(o);
```

Won't compile because Apple **doesn't** have:

```
compareTo(Orange o)
```

It only has:

```
compareTo(Apple o)
```

Fancier Generics

What if I care a little bit what T is?

```
SortableCollection<String> c= ...
c.sort();
```

requires T to be Comparable!

Fancier Generics

What if I care a little bit what T is?

```
SortableCollection<String> c= ...
c.sort();
```

requires T to be Comparable<T>!

```
interface SortableCollection<T extends Comparable<T>>
{
   ...
}
```

Two Slides on Exceptions: 1

- Exceptions make your code crash at **runtime**.
- You can catch them using a try/catch block:

```
try {
   // some code that might cause an error
} catch (TypeOfExceptionToCatch e) {
   // respond to the error in some sensible way
   // e points to the Exception that was thrown
}
```

Two Slides on Exceptions: 1

- Exceptions make your code crash at **runtime**.
- You can catch them using a try/catch block:

```
try {
   b = 10/0;
} catch (ArithmeticException e) {
   b = 0; // math works like this, right?

}
```

Two Slides on Exceptions: 2

- Sometimes your code **should** crash at runtime.
  - e.g., a precondition is violated
- You can force an exception - simply create an `Exception` and `throw` it:

```
if (bad_thing_happened) {
  throw new BadThingHappenedException();
}
```

Two Slides on Exceptions: 2

- Sometimes your code **should** crash at runtime.
  - e.g., a precondition is violated
- You can force an exception - simply create an `Exception` and `throw` it:

```
if (index > a.size) {
  throw new ArrayIndexOutOfBoundsException();
}
```

Two Slides on Exceptions: 2

- Sometimes your code **should** crash at runtime.
  - e.g., a precondition is violated
- You can force an exception - simply create an `Exception` and `throw` it:

```
// I haven't written this method yet
throw new UnsupportedOperationException();
```