Comparables, Autoboxing, and Exceptions

1 Comparables and Comparators

It's pretty straightforward on how to compare integers. Integers can be greater than, less than, or equal to each other. However, what if we wanted to compare different objects such as porcupines? There is no straightforward way that we can compare porcupines; you may want to compare them by how large they are, how many spines they have, or their names. In order to solve this problem, we can utilize comparables and comparators.
Comparable is an interface that can be implemented in a class. When implemented comparable, you must

write an *int compareTo(T o)* method. The Object that is passed in is what is being compared to the current instance. Frequently, the compareTo method will return a negative number if the current instance is less than the object passed in, a positive number if it's greater than the passed in instance, and 0 if the 2 objects are equal.

```
class FatCat<T> implements Comparable<T>{
    int weight;

    public FatCat(int weight) {
        this.weight = weight;
    }

    public int compareTo(T o) {
        if (!(o instanceof FatCat)) {
            return 1;
        }
        return this.weight - ((FatCat) o).weight;
    }
}
```

In the above example note how we have to check if the object passed in is an instance of Human. If we do not make this check, we may end up with a runtime error. Additionally, note how we need to cast the passed in object to be a human, if we do not do this, we will not be able to access the weight instance variable because the computer is not smart enough realize that the object is a Human.
Instead of comparables, we can use *comparators.* Comparators are also an interface that need to be imple-

mented, but instead of a *compareTo* method, they have an *int compare(T o1, T o2)* method. To modify the prior example, we can just modify the header to be

```
class FatCat implements Comparator<FatCat>
```

and the compareTo method would be replaced with

```
public int compare(FatCat o1, FatCat o2) {
        return o1.weight - o2.weight;
    }
```

We did pretty much the same thing that we did for comparables; however, if you notice, the header for our class has *Human* in it. What does this mean? It means that the generic object that Comparator passes in will be of Human type. This means that we no longer have to perform and *instanceof* check, as we know all of the objects passed in will be humans.

2 Autoboxing and widening

Autoboxing is a recent feature that was implemented in Java that allows for implicit conversions between wrappers and primitives- for example int to Integer and vice versa. One of the few times that autoboxing does not work is when we declare an array.

```
int[] testArr1 = new int[6];
int[] testArr2= new Integer[6]; //does not compile
```

Another concept that we will briefly touch on is the conversions between a smaller item to a larger. For example, a type int may be able to be passed in for a double argument, but a double will not be able to be passed in for an int. This can be thought of as "widening". This works because doubles have a larger range than ints. Think of it as trying to fit into a shirt- you can fit into a large shirt no problem, it'll just be baggy, but if you try to fit in a tiny shirt you may cause some problems.

3 Exceptions

If a program depends on some sort of user input, it is sometimes inevitable that we will have an error in our program. In order to minimize the damage that such an error does, we can use *try*, *catch*, and *finally*. The try statement allows us to run a program and if an exception is thrown that matches the exception a catch statement is looking for, we will move to the catch block. When an exception is caught, your program will no longer error out. After everything is completed, we visit the finally block. The finally block is very special because regardless of what happens, it will be run. If the program ends up erroring out due to an uncaught exception, finally will run right before exception is thrown. See if you understand what's going on in the code snippet below.

```
try {
    int[] arr = new int[3];
    int index = 0;
    while (true) {
        arr[index] = index;
        System.out.print(index);
      }
} catch(ArrayIndexOutOfBoundsException E){
          System.out.print("Phew got out of there");
} finally {
System.out.print("I need a nap");
 }
```

0 1 2 Phew got out of there I need a nap

Something important to note is that you must have the proper exception caught. If the above catch block was replaced with

```
catch(NullPointerException E){
     System.out.print("Wrong exception");
}
```

The exception would not get caught and we would instead have printed "0 1 2 I need a nap java.lang.arrayindexoutofboundsexce
For one try statement, you can have more than 1 catch block, this makes it so a variety of errors can be

caught.
There are two types of exceptions that can be thrown in a program, *checked exceptions* and *unchecked exceptions*. Checked exceptions are exceptions that need to be declared in the method header. The purpose of this is to prevent you from compiling your code so that you cannot have a runtime error. To declare that

you may be catching a checked exception, use the *throws* keyword. This is not to be confused with the *throw* keyword which is used to actually throw an exception. An example of throws and throw is below.

```
public static void oinkos() throws IOException{
...throw new IOException("protein power")...
}
```

Note throws will always be in the header of a method and throw will always be in the body of it.

Unchecked Exceptions, on the other hand, do not need to be declared in the method header. Unchecked exceptions crash at runtime, and are the exceptions that are typically used for our purposes.