Enumerated Types:

- Special kind of class
- Extends the enum class - therefore, it can only extend the java.lang.Enum class

Example:

```
public enum PizzaStatus {
ORDER,
READY,
DELIVER
}
```

Any enum class must have constants.

Enum classes can go in their own separate file or get embedded into other files.

We use enum classes so that we can only receive input that our program can handle

`Enum` is a reserved word in java.

Once the enum type is defined, you can declare a variable of that type and assign values.

```
PizzaStatus x = PizzaStatus.READY;
System.out.println(x); //READY
```

Example:

```
public enum Traffic {
GREEN, YELLOW, RED
}

//In some class
public void approachingLight(TrafficLight t) {
      if(t == TrafficLight.RED) {
            System.out.println("Stop!");
      }
}
```

Methods of the Enum Class:

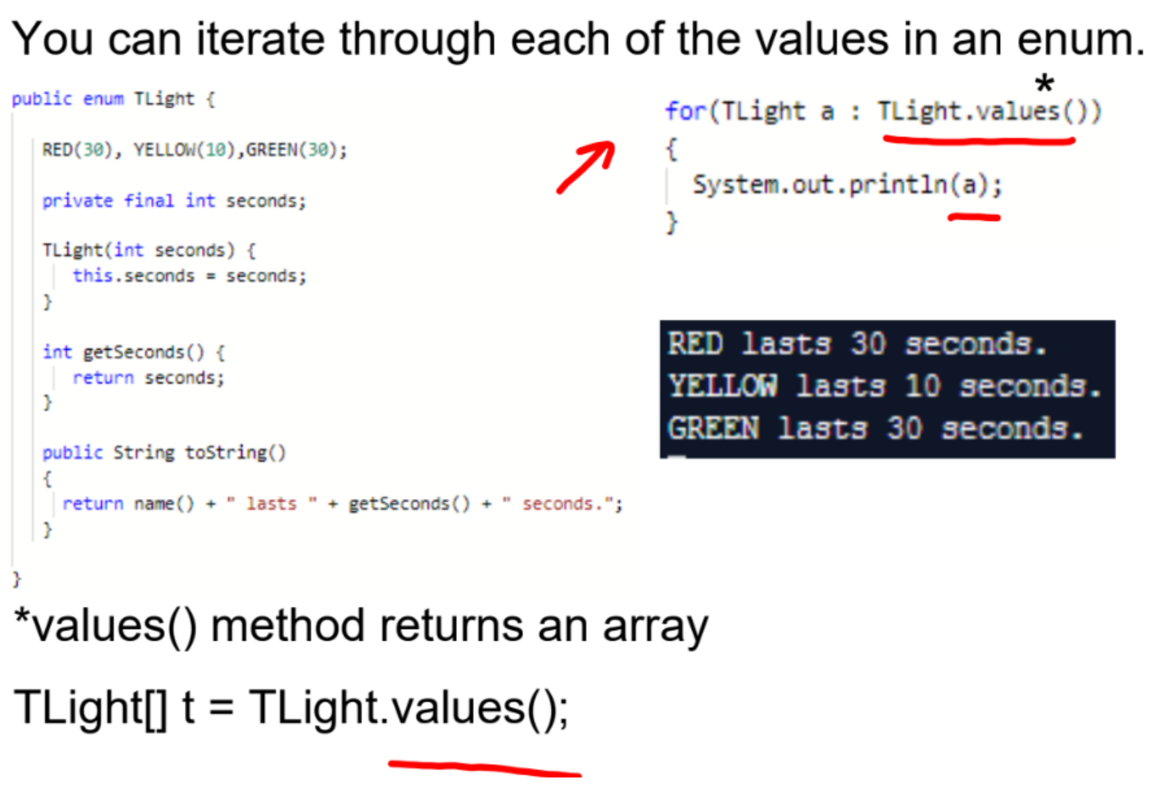
For a Enum class e

`public final int ordinal()` - returns an int representing the values position in the declaration

`public final string name()` - returns the names as it is in declaration

`public String toString()` - same as name, but can be overridden `public e[] values()` - returns an array of all the constants.

A constructor *cannot* be public



You can iterate through each of the values in an enum.

```
public enum TLight {

    RED(30), YELLOW(10),GREEN(30);

    private final int seconds;

    TLight(int seconds) {
        this.seconds = seconds;
    }

    int getSeconds() {
        return seconds;
    }

    public String toString()
    {
        return name() + " lasts " + getSeconds() + " seconds.";
    }

}
```

```
for(TLight a : TLight.values())
{
    System.out.println(a);
}
```

```
RED lasts 30 seconds.
YELLOW lasts 10 seconds.
GREEN lasts 30 seconds.
```

*values() method returns an array

TLight[] t = TLight.values();

You can add an abstract method to an enum

- Do this with `public abstract <name>`

For everyone's constant value you need to implement that abstract method within each declaration of an enum value.

Example:

```
public enum distances {
Novice(1.5) {
        public String method1() {
                return "short";
        }
}
GirlVarsity(2.5) {
        public String method1() {
                return "long";
        }
}
BoyVarsity(3.1) {
        public String method1() {
                return "very long";
```

```
      }

}
private int miles;

distances(int miles) {
this.miles = miles;
}

public abstract String method1(){ }
}
```

Control Statements (Do-While, Switch Case):

Do-While Statements:

```
do {
//code
} while(condition)
```

Literally just a while loop.
Except the code must run at **least once.** In a while loop, it is possible that the inner code runs 0 times.

Switch Statement:

A control structure that allows us to replace several nested if-else constructs:

```
if(animal.equals("DOG") || animal.equals("CAT")) {
System.out.println("domestic");
}
else if(animal.equals("TIGER")) {
System.out.println("wild");
}
else  {
System.out.println("unknown");
}
```

In Switch-case

```
switch(animal) {
      case "DOG":
            System.out.println("domestic");
            break;
      case "CAT":
            System.out.println("domestic");
            break;
      case "TIGER":
            System.out.println("wild");
            break;
      default:
            System.out.println("unknown");
            break;

}
```
Don't forget to use the word "break".
If you don't use the word break, and the switch statement finds a match, all the codeblocks underneath will execute until the computer finds a break.

<u>More Efficient Way:</u>
```
switch(animal) {
      case "DOG":
/*
notice how the lack of break means the computer will "fall through"
and keep executing all the blocks of code below until it finds
"break". This includes falling through default.
*/
      case "CAT":
            System.out.println("domestic");
            break;
      case "TIGER":
            System.out.println("wild");
            break;
      default:
            System.out.println("unknown");

}
```

When you put variables in the case, make sure those variables are final.

`[variable]--` vs `--[variable]`

- (Post increment)First one uses the variable in the operation, the decrements
- (Pre increment) Second one decrements first

Example 1:
```
int x = 34;
int y = ++x; // both x and y are 35
```
Example 2:
```
int x = 34;
int y = x++; //y is 34, x is 35
```

Ternary Operator:

It is the “?” symbol.
Can be used to replace a simple if-else statement.

```
Boolean condition ? expression 1: expression 2
```

Example:

`int a;`
`if(x > y) {`
`a = 1;`
`}`
`else {`
`a = 2;`
`}`
	`int a = x > y ? 1: 2`

This produces the same output.
The expressions must be some kind of value that can be returned.

Exceptions:

An exception is a class used to store error information.
Each exception is an object (instance) of the class.

Two parts of dealing with program error:
Detection of the error
Handling of errors.

Example:

The Scanner class can read a file object. A possible error is the Scanner tries to read a file that does not exist.

- *Detection: Scanner cannot find file*
- *Handle: the program needs to deal with this issue.*

We can “throw” exceptions when you detect an error condition.
Example: *you try to withdraw more money than is in the bank account:*

```
if (amount > balance) { //detection
      //Handle it somehow. For example,
      throw new IllegalArgumentException(“Amount exceeds balance”)
}
```

Notice how you use the reserved word `throw` followed by some `ExceptionObject`.

You can create your own exception subclasses!

Try/Catch:

A way to handle exceptions and try to use code that might cause errors.
Three reserved words:
`try` - computer tries to run this code.
`catch` - computer does this if try code causes error.

- There can be multiple catch blocks.

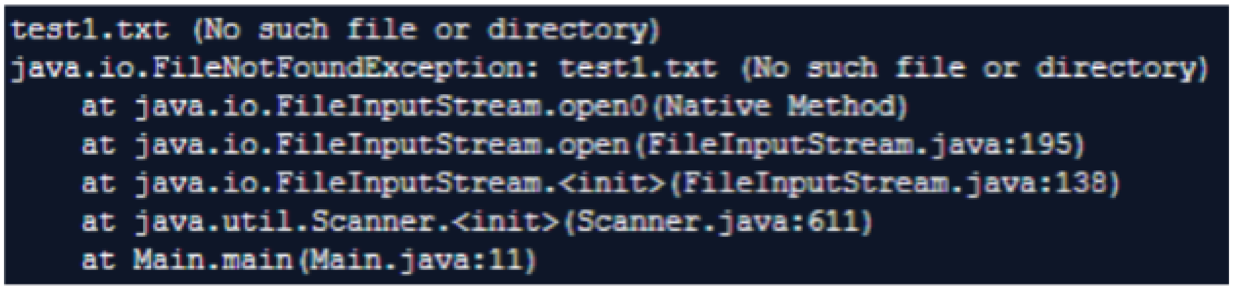
`finally` - this code happens no matter what

Example:
```
try {
      Scanner scan = new Scanner(new File("test.txt"));
}
catch(FileNotFoundException e) {
      System.out.println(e.getMessage())
}
//more code
```
Importantly, this program will continue to run even after the try/catch error. It will not just terminate because of the error - the rest of the program will run.

Because the variable is declared within the `try {}`, it only exists there. The scope means it cannot be used outside the try block.

To get around this, do:
```
Scanner scan = null;
try {
      scan = new Scanner(new File("test.txt"));
}
catch(FilenotFoundException e) {
      System.out.println(e.getMessage())
}
//more code
```

Method: The printStackTrace() method shows the chain of method calls that lead to this exception
Example:



```
test1.txt (No such file or directory)
java.io.FileNotFoundException: test1.txt (No such file or directory)
    at java.io.FileInputStream.open0(Native Method)
    at java.io.FileInputStream.open(FileInputStream.java:195)
    at java.io.FileInputStream.<init>(FileInputStream.java:138)
    at java.util.Scanner.<init>(Scanner.java:611)
    at Main.main(Main.java:11)
```

Error:

Three types:
Error - internal errors like OutOfMemory, fatal error. Not discussed in this class
Unchecked Exceptions - Descendants of RunTimeException

- These may occur while a program is running - not your fault

Checked Exception - all other exceptions

- The computer will not let you run the program unless you include code to deal with possible errors.

Comparable Interface:

Looks like this:
```
public interface Comparable<T>
{
public abstract int compareTo(T o) {}
}
```

T is a generic placeholder for the type of object being compared.

Know that the compareTo method *must* throw a `NullPointerException`.

The comparable interface and the compareTo method can aid in sorting.
For example, if we want to sort ArrayList, we would need to use `Collections.sort()` from `import java.util.Collections;`. This method used the Comparable interface.

Some rules that must be followed when implementing a compareTo method:

1. sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
2. x.compareTo(y) > 0 && y.compareTo(z) > 0 implies x.compareTo(z) > 0.
3. (x.compareTo(y) == 0) implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z))
4. **(x.compareTo(y)==0) == x.equals(y) //Important

Example:
Can be useful if you need to sort an Object based on multiple criteria.

```
 ArrayList<Student> WHS = new ArrayList<Student>();
        for(double i = 2; i < 4; i++) {
            for(int j = 0; j < 3; j++) {
                WHS.add(new Student(i,j));
            }
        }
        System.out.println(WHS);
        Collections.sort(WHS);
        System.out.println(WHS);
    }
    private static class Student implements Comparable<Student> {
        private double GPA;
        private int sports;
        public Student(double GPA, int sports) {
            this.GPA = GPA;
            this.sports = sports;
        }
        public int compareTo(Student other) {
            if(Integer.compare(sports,other.sports) != 0) {
                return Integer.compare(sports, other.sports);
            }
            return Double.compare(GPA, other.GPA);
        }
        public String toString() {
            return "Sports: " + sports + " GPA: " + GPA;
        }
    }
```

This sorts students in ascending order. It prioritizes sports first - if two students have the same number of sports, only then will it sort in order of GPA.

Notice the wrapper classes of primitive types like int and double have methods (compare()) to help override the abstract compareTo method.

Output:

```
[Sports: 0 GPA: 2.0, Sports: 1 GPA: 2.0, Sports: 2 GPA: 2.0, Sports: 0 GPA: 3.0, Sports: 1 GPA: 3.0, Sports: 2 GPA: 3.0]
[Sports: 0 GPA: 2.0, Sports: 0 GPA: 3.0, Sports: 1 GPA: 2.0, Sports: 1 GPA: 3.0, Sports: 2 GPA: 2.0, Sports: 2 GPA: 3.0]
```

.equals() and .hashCode():

`.hashCode()` returns an integer value, generated by a hashing algorithm (returns a random number that represents all the stuff about the object it's called on).

Objects that are equal (according to their `.equals()`) must return the same hash code. Different objects do not need to return different hash codes.

Variable number of Arguments (Varags)

A method that can take in a variable number of arguments. The method turns all the parameters and turns it into an array.
Use "…" to do this.

Example: (Pretend this is in the main class)

```
public static int test(int…x) {
      int sum = 0;
      for(int y: x) {
            sum += y;
      }
      return sum;
}
```

Valid method calls could include:

```
Main.test(1,2,3,4); //returns 10
Main.test(); //returns 0
Main.test(10,15); //returns 25
```

Comparator Interface:

The Comparator<T> interface allows us to define how to make comparisons and possibly use parameters to change who the comparison is made.

```
Public interface Comparator<T> {

      public abstract int compare(T o1, T o2);
}
```

This interface is not implemented in the class of object "T", which is why it requires two parameters. It is implemented in some other class.

A proper compare method should obey the same rules as the compareTo method.

1. sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
2. x.compareTo(y) > 0 && y.compareTo(z) > 0 implies x.compareTo(z) > 0.
3. (x.compareTo(y) == 0) implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z))
4. **(x.compareTo(y)==0) == x.equals(y) //Important