You first write something like this:

```java
enum TokenType implements IEnumVal<String> {
    NUMBER("\d+"),
    PLUS("\+"),
    TIMES("\*"),
    MINUS("-"),
    DIVIDE("/"),
    EQUALS("="),
    WHITESPACE("[\s]+"),
    FAIL(""); // if the matcher fails, you get one of these
}
```

What's an enum?
Old school: “final” constants

We did this in the Log class

```java
public final class Log {
    private static final String TAG = "edu.rice.Log";

    /**
     * logging level: everything goes into the log
     */
    public final static int ALL = 2;

    /**
     * logging level: only warning and error go into the log
     */
    public final static int WARNING = 1;

    /**
     * logging level: only errors go into the log
     */
    public final static int ERROR = 0;

    /**
     * logging level: nothing is logged at all
     */
    public final static int NOTHING = -1;

    private static int logLevel = ALL;
}
```
We did this in the Log class

```java
public final class Log {
    private static final String TAG = "edu.rice.Log";

    /**
     * logging level: everything goes into the log
     */
    public final static int ALL = 2;

    /**
     * logging level: only warning and error go into the log
     */
    public final static int WARNING = 1;

    /**
     * logging level: only errors go into the log
     */
    public final static int ERROR = 0;

    /**
     * logging level: nothing is logged at all
     */
    public final static int NOTHING = -1;

    private static int logLevel = ALL;
}
```

What if some other integer arrived? Java won’t automatically reject it.
New school: "enum" types

```java
public class Car {
    public enum Model {
        SPARK, SONIC, CRUZE, MALIBU, IMPALA, VOLT, SS, CAMARO, CORVETTE, TRAX, EQUINOX, TRAVERSE, TAHOE, SUBURBAN, COLORADO, SILVERADO, EXPRESS, CITY_EXPRESS
    }

    public enum Color {
        WHITE, BLACK, RED, SILVER, GREY, BLUE, LIMEGREEN
    }

    public final Model model;
    public final Color color;

    public Car(Model model, Color color) {
        this.model = model;
        this.color = color;
    }

    public final static Car specialVette = new Car(Model.CORVETTE, Color.LIMEGREEN);
}
```
New school: “enum” types

```java
public class Car {
    public enum Model {
        SPARK, SONIC, CRUZE, MALIBU, IMPALA, VOLT, SS, CAMARO, CORVETTE,
        TRAX, EQUINOX, TRAVERSE, TAHOE, SUBURBAN,
        COLORADO, SILVERADO, EXPRESS, CITY_EXPRESS
    }

    public enum Color {
        WHITE, BLACK, RED, SILVER, GREY, BLUE, LIMEGREEN
    }

    public final Model model;
    public final Color color;

    public Car(Model model, Color color) {
        this.model = model;
        this.color = color;
    }

    public final static Car specialVette = new Car(Model.CORVETTE, Color.LIMEGREEN);
}
```

Optional: “public” makes the enum visible outside of the Car class
New school: “enum” types

```java
public class Car {
    public enum Model {
        SPARK, SONIC, CRUZE, MALIBU, IMPALA, VOLT, SS, CAMARO, CORVETTE,
        TRAX, EQUINOX, TRAVERSE, TAHOE, SUBURBAN,
        COLORADO, SILVERADO, EXPRESS,
    }

    public enum Color {
        WHITE, BLACK, RED, SILVER, GREY, BLUE, LIMEGREEN
    }

    public final Model model;
    public final Color color;

    public Car(Model model, Color color) {
        this.model = model;
        this.color = color;
    }

    public final static Car specialVette = new Car(Model.CORVETTE, Color.LIMEGREEN);
}
```

When you’re outside of Car, you’d need to say Car.Color.LIMEGREEN
What’s going on under the hood?

Enums are just Java classes, with exactly one instance per value

```java
define Color {
    WHITE, BLACK, RED, SILVER, GREY, BLUE, LIMEGREEN
}
```

How many instances of Color? Exactly seven.

If you have two Colors, you can compare them with `==`

Runs fast, just testing pointer equality

But wait, there’s more!
Every enum defines some useful methods

String name();
Returns literally the textual name of the value ("WHITE", "BLACK", "RED", ...)

int compareTo(E anotherEnum);
int ordinal();
 Enums are “ordered”, just like we did with our constants in the Log

Car.Color color = Enum.valueOf(Car.Color.class, "RED");
If you have the string, there’s a static method to go back to the enum
Every enum defines some useful methods

String name();
Returns literally the textual name of the value ("WHITE", "BLACK", "RED", ...)

int compareTo(E anotherEnum);
int ordinal();
Enums are “ordered”, just like we did with our constants in the Log

Car.Color color = Enum.valueOf(Car.Color.class, "RED");
If you have the string, there's a static method to go back to the enum

What's this .class business about?
Coming in a few slides.
Enum constructors

You can pass arguments to an enum constructor and use them later

```java
enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
    Coin(int value) { this.value = value; }

    private final int value;
    public int value() { return value; }
}
```
Enum constructors

You can pass arguments to an enum constructor and use them later

```java
class Coin {
    enum {
        PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
        Coin(int value) {
            this.value = value;
        }
    }
    private final int value;
    public int value() {
        return value;
    }
}
```
Enum constructors

You can pass arguments to an enum constructor and use them later

```java
enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
    Coin(int value) { this.value = value; }

    private final int value;
    public int value() { return value; }
}
```

Add more methods? No problem.
Enum interfaces / inheritance

Enums are *final*. Not allowed to extend other classes.
But it’s fine to implement an interface!

```java
enum SimpleTokenTypes implements IEnumVal<String> {
    OPENCURLY("\{"),
    CLOSECURLY("\}"),
    WHITESPACE("\s"),
    EQUALS("="),
    SEMICOLON(";"),
    WORD("\p{Alnum}+"),
    FAIL("");

    public final String pattern;

    SimpleTokenTypes(String pattern) { this.pattern = pattern; }

    // required by NamedMatcher
    public String value() { return pattern; }
}
```
But what’s up with the .class thing?

We’re working around a limitation in Java.

Several of you have sometimes tried to do something like:

```java
public class Whatever<T> {
    T getValue() {
        T something = new T();
        return something;
    }
}
```
But what’s up with the .class thing?

We’re working around a limitation in Java.

Several of you have sometimes tried to do something like:

```java
public class Whatever<T> {
  T getValue() {
    T something = new T();
    return something;
  }
}
```

Forbidden! (Because “type erasure”, mentioned in week 1.)
public class Whatever<T> {
    Whatever(Class<T> clazz) {
        this.clazz = clazz;
    }

    private Class<T> clazz;

    T getValue() {
        try {
            T something = clazz.newInstance();
            return something;
        } catch (InstantiationException | IllegalAccessException e) {
            e.printStackTrace();
            return somethingElse;
        }
    }
}
public class Whatever<T> {
    Whatever(Class<T> clazz) {
        this.clazz = clazz;
    }

    private Class<T> clazz;

    T getValue() {
        try {
            T something = clazz.newInstance();
            return something;
        } catch (InstantiationException | IllegalAccessException e) {
            e.printStackTrace();
            return somethingElse;
        }
    }
}
Pointers to a Java class. Itself!

```java
public class Whatever<T> {
    Whatever(Class<T> clazz) {
        this.clazz = clazz;
    }

    private Class<T> clazz;

    T getValue() {
        try {
            T something = clazz.newInstance();
            return something;
        } catch (InstantiationException | IllegalAccessException e) {
            e.printStackTrace();
            return somethingElse;
        }
    }
}
```

Class has a type parameter (weird, huh?) used to ensure it matches T
Pointers to a Java class. Itself!

```java
public class Whatever<T> {
    Whatever(Class<T> clazz) {
        this.clazz = clazz;
    }

    private Class<T> clazz;

    T getValue() {
        try {
            T something = clazz.newInstance();
            return something;
        } catch (InstantiationException | IllegalAccessException e) {
            e.printStackTrace();
            return somethingElse;
        }
    }
}
```

Lots of strange new methods you can do!
“Reflection”: APIs to interrogate objects

Once you have a Class object, there are all kinds of weird methods
Example: Given a class, get a list of all its constructors
And call them if you want!

Or get a list of all the methods and fields on an object

You can even see all those @annotations

For the most part, you won’t be using Java’s reflection features at all
But some APIs use it, so you need to have a passing understanding
I wanted a really simple API
Just write the regular expressions for each token in an enum
And interrogate it for all the values

I needed a way to "pass the enum" to the NamedMatcher
Java reflection to the rescue!
NamedMatcher's use of reflection

```java
public interface IEnumVal<T> {
    String name(); // this comes from the enum
    T value(); // you have to implement this on
}

class NamedMatcher<T extends Enum<T> & IEnumVal<String>> {
    ...

    public NamedMatcher(Class<T> enumClazz) {
        if (!enumClazz.isEnum())
            throw new RuntimeException("NamedMatcher requires an enum class");

        // this gets us an array of all the enum values in the type.
        T[] enumConstants = enumClazz.getEnumConstants();
    }
```
NamedMatcher's use of reflection

```java
public interface IEnumVal<T> {
    String name(); // this comes from the enum
    T value(); // you have to implement this on
}

class NamedMatcher<T extends Enum<T> & IEnumVal<String>> {
...

    public NamedMatcher(Class<T> enumClazz) {
        if (!enumClazz.isEnum())
            throw new RuntimeException("NamedMatcher requires an enum class");

        // this gets us an array of all the enum values in the type.
        T[] enumConstants = enumClazz.getEnumConstants();

        Type constraint: T must be an Enum
```
NamedMatcher’s use of reflection

```java
public interface IEnumVal<T> {
    String name(); // this comes from the enum
    T value(); // you have to implement this on
}

class NamedMatcher<T extends Enum<T> & IEnumVal<String>> {
    ...

    public NamedMatcher(Class<T> enumClazz) {
        if(!enumClazz.isEnum())
            throw new RuntimeException("NamedMatcher requires an enum class");

        // this gets us an array of all the enum values in the type.
        T[] enumConstants = enumClazz.getEnumConstants();
    }
}
```

And it needs a “value” method.
public interface IEnumVal<T> {
    String name(); // this comes from the enum
    T value(); // you have to implement this
}

class NamedMatcher<T extends Enum<T> & IEnumVal<String>> {
    ...

    public NamedMatcher(Class<T> enumClazz) {
        if(!enumClazz.isEnum())
            throw new RuntimeException("NamedMatcher requires an enum class");

        // this gets us an array of all the enum values in the type.
        T[] enumConstants = enumClazz.getEnumConstants();

        Not just any Class; constrained to be an Enum and to implement IEnumVal
public interface IEnumVal<T> {
    String name(); // this comes from the enum
    T value(); // you have to implement this
}

class NamedMatcher<T extends Enum<T> & IEnumVal<T>> {
    ...

    public NamedMatcher(Class<T> enumClazz) {
        if (!enumClazz.isEnum())
            throw new RuntimeException("NamedMatcher requires an enum class");

        // this gets us an array of all the enum values in the type.
        T[] enumConstants = enumClazz.getEnumConstants();
    }
Could we reengineer Tree & Treap?

You wrote almost exactly the same thing in some cases, e.g.:  

Tree<T> rotateRight() {
    if(left.empty()) return this;
    return new Tree<>
        (left.getValue(),
         left.getLeft(),
         new Tree<>
        (value, left.getRight(), right));
}

Treap<T> rotateRight() {
    if(left.empty()) return this;
    return new Treap<>
        (left.getValue(),
         left.getLeft(),
         new Treap<>
        (value, left.getRight(), right, priority), left.getPriority());
}

Problem 1: Reflection (e.g., trying to use .newInstance() here) is slow.
Problem 2: Where do you store those Class instances themselves? Once per tree node? Serious memory overhead.

Consensus: Use reflection capabilities when you have no other choice or when performance is irrelevant.
Mockito is a spectacularly useful library that complements JUnit and it uses Java’s reflection capabilities all over the place.

The broad idea is simple
Create “mock” versions of classes that you want to test
They use the “real” interface but have none of your code inside
You tell them how to respond
You then do your tests

All kinds of uses
Example: make a fake network that always sends you the same message
Installing Mockito

Hopefully already there. If not, IntelliJ can install the library from Maven
org.mockito:mockito-all:1.10.19

Then just:
import static org.mockito.Mockito.*;
Example 1: Verifying LazyList laziness

See testLaziness in LazyListTest.java

Supplier<IList<String>> supplier = mock(Supplier.class);
when(supplier.get()).thenReturn(new LazyList<>()("Eve"); // eavesdropping

IList<String> lazy1 = new LazyList<>()("Dorothy", supplier);
IList<String> lazy = LazyList.of("Alice", "Bob", "Charlie");
IList<String> longerList = lazy.concat(lazy1);
verify(supplier, never()).get(); // otherwise, the concat screwed up
IList<String> ucaseList = longerList.map(String::toUpperCase);

// okay, now we're going to read the elements from the lazy list in series
assertEquals("ALICE", ucaseList.head());
verify(supplier, never()).get();
assertEquals("BOB", ucaseList.tail().head());
verify(supplier, never()).get();
assertEquals("CHARLIE", ucaseList.tail().tail().head());
verify(supplier, never()).get();
assertEquals("DOROTHY", ucaseList.tail().tail().tail().head());
verify(supplier, never()).get();
assertEquals("EVE", ucaseList.tail().tail().tail().tail().head());
verify(supplier, atLeastOnce()).get();
Example 1: Verifying LazyList laziness

See testLaziness in LazyListTest.java

```java
Supplier<IList<String>> supplier = mock(Supplier.class);
when(supplier.get()).thenReturn(new LazyList<>("Eve"); // eavesdropping

IList<String> lazy1 = new LazyList<>("Dorothy", supplier);
IList<String> lazy = LazyList.of("Alice", "Bob", "Charlie");
IList<String> longerList = lazy.concat(lazy1);
verify(supplier, never()).get(); // otherwise, the concat screwed up
IList<String> ucaseList = longerList.map(String::toUpperCase);

// okay, now we're going to read the elements from the lazy list in series
assertEquals("ALICE", ucaseList.head());
verify(supplier, never()).get();
assertEquals("BOB", ucaseList.tail().head());
verify(supplier, never()).get();
assertEquals("CHARLIE", ucaseList.tail().tail().head());
verify(supplier, never()).get();
assertEquals("DOROTHY", ucaseList.tail().tail().tail().head());
verify(supplier, never()).get();
assertEquals("EVE", ucaseList.tail().tail().tail().tail().head());
verify(supplier, atLeastOnce()).get();
```

Normal LazyLists: Suppliers are lambdas that supply the tail values. Not here!
Example 1: Verifying LazyList laziness

See testLaziness in LazyListTest.java

```java
Supplier<IList<String>> supplier = mock(Supplier.class);
when(supplier.get()).thenReturn(new LazyList<>("Eve")); // eavesdropping

IList<String> lazy1 = new LazyList<>("Dorothy", supplier);
IList<String> lazy = LazyList.of("Alice", "Bob", "Charlie");
IList<String> longerList = lazy.concat(lazy1);
verify(supplier, never()).get(); // otherwise, the concat screwed up
IList<String> ucaseList = longerList.map(String::toUpperCase);

// okay, now we're going to read the elements from the lazy list in series
assertEquals("ALICE", ucaseList.head());
verify(supplier, never()).get();
assertEquals("BOB", ucaseList.tail().head());
verify(supplier, never()).get();
assertEquals("CHARLIE", ucaseList.tail().tail().head());
verify(supplier, never()).get();
assertEquals("DOROTHY", ucaseList.tail().tail().tail().head());
verify(supplier, never()).get();
assertEquals("EVE", ucaseList.tail().tail().tail().tail().head());
verify(supplier, atLeastOnce()).get();
```

When the (mock) lambda is invoked, return the next step in the LazyList.
Example 1: Verifying LazyList laziness

See testLaziness in LazyListTest.java

```
Supplier<IList<String>> supplier = mock(Supplier.class);
when(supplier.get()).thenReturn(new LazyList<>("Eve"); // eavesdropping

IList<String> lazy1 = new LazyList("Dorothy", supplier);
IList<String> lazy = LazyList.of("Alice", "Bob", "Charlie");
IList<String> longerList = lazy.concat(lazy1);
verify(supplier, never()).get(); // otherwise, the concat screwed up

IList<String> ucaseList = longerList.map(String::toUpperCase);

// okay, now we're going to read the elements from the lazy list in series
assertEquals("ALICE", ucaseList.head());
verify(supplier, never()).get();
assertEquals("BOB", ucaseList.tail().head());
verify(supplier, never()).get();
assertEquals("CHARLIE", ucaseList.tail().tail().head());
verify(supplier, never()).get();
assertEquals("DOROTHY", ucaseList.tail().tail().tail().head());
verify(supplier, never()).get();
assertEquals("EVE", ucaseList.tail().tail().tail().tail().head());
verify(supplier, atLeastOnce()).get();
```

At each step of reading the LazyList, we can verify that the Supplier was never called.
Couldn’t I have done that without Mockito?

Yes. You could have made a Supplier that mutated some state:

```java
class Tripwire {
    public boolean tripped = false;
}

@Test
public void testLaziness2() throws Exception {
    Tripwire tripwire = new Tripwire();

    Supplier<IList<String>> supplier = () -> {
        tripwire.tripped = true;
        return new LazyList<>("Eve");
    };

    IList<String> lazy1 = new LazyList<>("Dorothy", supplier);
    IList<String> lazy = LazyList.of("Alice", "Bob", "Charlie");
    IList<String> longerList = lazy.concat(lazy1);
    ...

    assertTrue(!tripwire.tripped);
}
Couldn’t I have done that without Mockito?

Yes. You could have made a Supplier that mutated some state:

class Tripwire {
    public boolean tripped = false;
}

@Test
public void testLaziness2() throws Exception {
    Tripwire tripwire = new Tripwire();

    Supplier<IList<String>> supplier = () -> {
        tripwire.tripped = true;
        return new LazyList<>("Eve");
    };

    IList<String> lazy1 = new LazyList<>("Dorothy", supplier);
    IList<String> lazy = LazyList.of("Alice", "Bob", "Charlie");
    IList<String> longerList = lazy.concat(lazy1);

    ...  
    assertTrue(!tripwire.tripped);
When to use Mockito then?

Read the documentation:
http://mockito.org/
https://dzone.com/refcardz/mockito
http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html

Ask yourself: where might I drop in a “mock” object instead of a real one?
Treap: mock left/right child treaps with fixed priorities ... test rebalancing
NamedMatcher: mock version returns fixed tokens, tests downstream code
Any sort of user interaction: mock version returns fixed values

Advanced topics: “spy” mocks
“Real” object inside, lets you make assertions about its behavior
Warning: performance might suck when testing
Coverage testing

The holy grail: your test cases “exercise” every line of code you’ve written
Still not necessarily bug free, but it’s an important property

IntelliJ has very nice support for this
If you don’t see “Run ... with Coverage” then you have a problem
Coverage results?

Tell you how your test cases are doing at exercising your code
This is good but not spectacular. Ideally, you want to hit all methods, most lines.
Test cases covered the constructor, but not the internal treap extraction. Should we test it? Yes! Or, do we even really need it? Maybe not.
Example: TreapSet

Test cases covered the constructor, but not the internal treap extraction
Should we test it? Yes! Or, do we even really need it? Maybe not.

Test coverage.

Click in the gutter and it tells you the “hit count”.

No test coverage.
Live coding exercise

Writing more unit tests to improve our test coverage