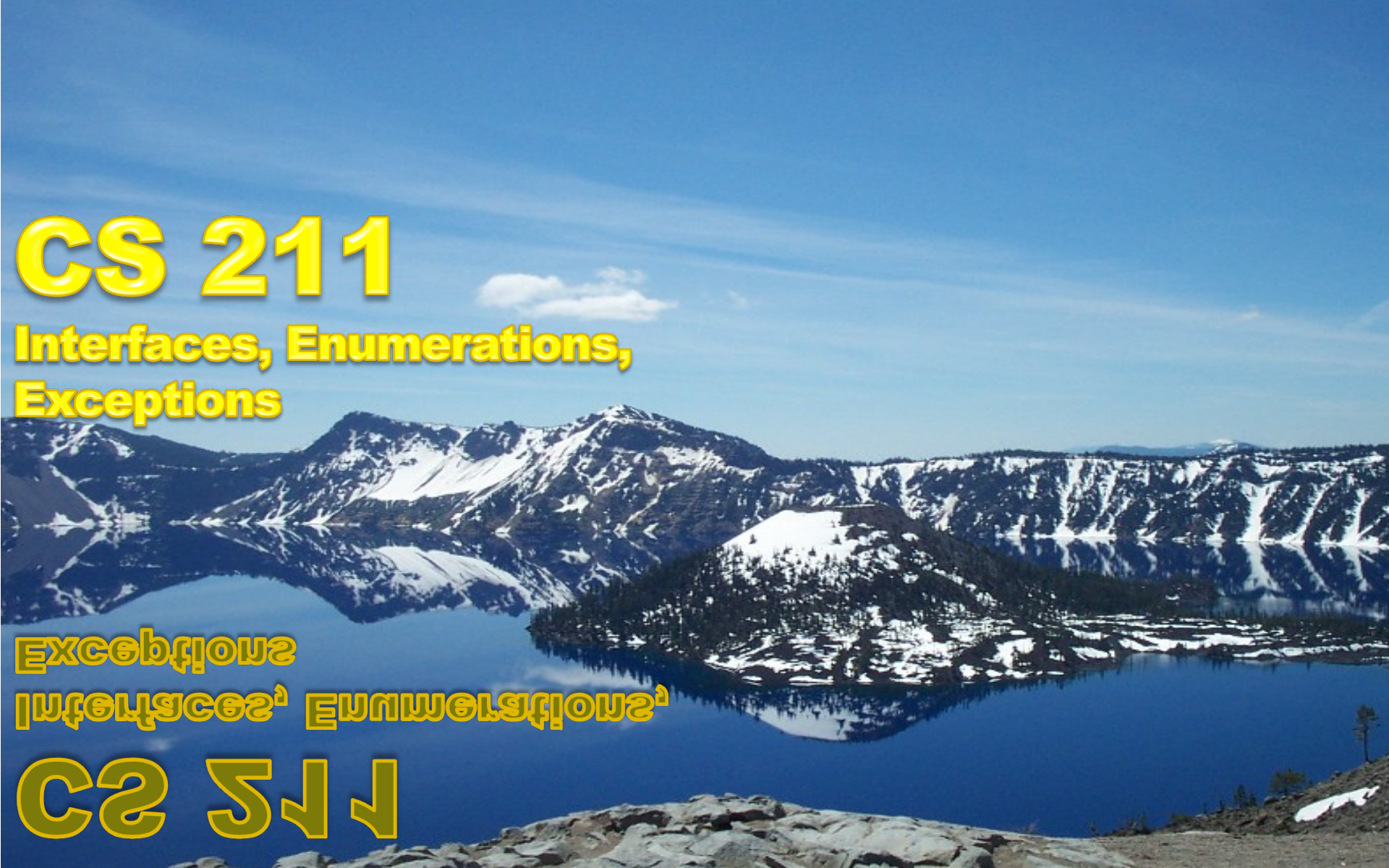
CS 211
Interfaces, Enumerations,
Exceptions

Interfaces

What Need Does an Interface Address?

- Java disallows *multiple inheritance*
- we always have one parent class
- How do we relate unrelated things?
  - example: our grocery store program wants to sell grapes, gum, batteries, and dog food.

→ How can we retrieve the price & barcode info for each?

→ We want an array of all these sellable items in the cart.

→ The Object class only sort of helps.

Interface: Introduction

interface: a set of **abstract** methods. (no fields are allowed)*

An **Interface** is a **Type**!

Any class can **implement** the interface:
→ claim they do, then implement all the abstract methods.

A single class can implement multiple interfaces by implementing all the methods of all the targeted interfaces.

there's actually much more to interfaces; this is just our first view of them!

Interface Example

```
public interface Sellable {
    public abstract int getPrice();   //in pennies
    public abstract int getBarCode();//unique int
    public abstract String receiptLine(int quantity);
 }
```

First, create an interface, declaring what methods must be implemented to be able to behave like this new type of thing.

We see three **abstract methods**: method signatures with no body.

→ We briefly introduced abstract methods for abstract classes. These are the same thing, and can similarly be overridden with new implementations.

Practice Problem

Create the `Noisy` interface, which has the `beLoud` method.

What classes might `implement Noisy`?

Implementing an Interface

A class declares that it implements the interface, and overrides the acquired methods

```
public class Battery extends EnergySource implements Sellable {
  // fields of the Battery class
  public int voltage;
  public String sizeDescriptor;

  //constructor, other methods in the class.
  public Battery (int v, String sd) { ... }

  //implementations for all Sellable methods.
  public int getPrice() { return 150; }
  public int getBarCode(){ return 1596783; }
  public String receiptLine(int qty) {
    int total = getPrice()* qty;
    return qty + " batteries @ $"+getPrice()+"......$"+ total;
  }
}
```

Practice Problems

Implement the `Sellable` interface for `DogFood`.

Implement your `Noisy` interface for any class of your choosing.

Using Interfaces

Use the interface type to refer to any object whose class implements it.

```
public static void main (String[] args) {
  Grape gr = new Grape(2,"red");
  DogFood df = new DogFood(35,"Old Yeller");

  //like superclasses, we can assign any Sellable thing to a Sellable variable.
  Sellable s = gr;

  //in the following, s, gr, df could be used in each others' places
  System.out.println("buying grapes? "+s.receiptLine(5));
  priceCheck(df);
  priceCheck(gr);
}

public static void priceCheck(Sellable sell) {
  System.out.println("checking price for "sell+":");
  System.out.println(sell.receiptLine(1));
}
```

Example

```
interface I { … }
class A implements I { … }
class B implements I { ... }
```

```
// in a method somewhere…
A a = new A();
B b = new B();
I i;  // we can have variables of type I.
i=a; // allowed: things of type A guarantee everything we need of type I.
i=b; // same reasoning.
b=i; // FAILS: type B may have extra def'ns not guaranteed by I.
b = (B) i; // downcast succeeds: actually points to B object
```

Examples of Interfaces

java.lang.Comparable. one method:

```
public int compareTo(Object other);
```

→ return value: negative:"less than" zero:"equal". positive:"greater than"
→ gives a consistent way to sort data.

Example: class String implements Comparable.
(`compareTo` is available on Strings)

---

java.lang.Iterator. Three methods:

```
public boolean hasNext();
public Object  next   ();
public void    remove ();
```

More Interfaces

`Serializable` interface: no methods!

- indicates that the object can be transformed into a byte sequence
- (for file storage, network transfer, etc).
- Serializable classes' objects can be stored in files!

Interface Inheritance

- we can **extend** one interface to create another. Child interface inherits all methods from parent interface
- multiple inheritance of interfaces is allowed!
- A good example of this is Java's Collections Library, provided as a hierarchical series of interfaces, also using the extends keyword.

Pytania Poll

- Interfaces

Enumerations

Enumerations

Purpose: list out all of the values in a finite set.

- examples: days of week; planets; grades.
- more error-proof than just using `int`s or `String`s: less chances to abuse the value.

Implementation Detail:

- Java uses the class mechanism behind the scenes.

→ an enumeration is "syntactic sugar" for a special usage of classes.

Enumeration – Creation

```
public enum Grade { A, B, C, D, F }
```

- Place enumeration in its own file, like a class.
- File name matches enum name. (e.g. `Grade.java`)

- An **enumeration** creates a **new type**.
- The values of the enumeration are … enumerated. (Listed explicitly).
- It looks like set notation in math!

Enumeration – Usage

- Direct usage of the values: `EnumName.EnumValue` `Grade.A`
- switch usage is allowed directly on enum value names:

```
//directly access the enum fields
Grade g = Grade.A;
System.out.println("The grade is " + g);

//notice that we DON'T say Grade.A, just A, inside a switch:
switch (g) {
  case A:
    System.out.println("Ace!!!!");
    break;
  case B:
    System.out.println("Buzz...");
     break;
  default:
    System.out.println("Meh....");
}
```

Iterators from Enums

- each enum has a **values()** method, giving an array of the values in order.
- use it with for-each loops.

```
for (Grade g : Grade.values()) {
     System.out.println("Grade: " + g);
}
```

Practice Problems

- Create an enum for the sign of a number (positive, negative, zero)
- Store a Sign to a variable. Use it in a switch to print out if the number is "bigger than", "equal to" , or "less than" zero.

- Create the Quadrant enumeration, representing the four quadrants.

Advanced Enumerators

adding fields and (non-public) constructors

```
public enum Day {
  // we add special constructor calls to the enumerated values.
  MON ("Monday",false),    TUES("Tuesday",false),  WED("Wednesday",false),
  THURS("Thursday",false), FRI("Friday",false),    SAT("Saturday",true),
  SUN("Sunday",true)  ;

  //we can create fields. (private just for encapsulation reasons)
  private String fullDesc;
  private boolean isWeekend;

  //(private/package-private only) constructor, called above.
  private Day (String fullDesc, boolean isWeekend){
    this.fullDesc = fullDesc;
    this.isWeekend = isWeekend;
  }
}
```

Advanced Enumerators

- Adding other methods (just like adding more methods to a class)
- This is basically a class now.

```
// being implemented as a class, we can also provide toString().
 public String toString(){
   return "<Day: "+fullDesc+"/ is"+(isWeekend?"n't":"")+"
weekday>";
 }

 //other public methods are also allowed...
 public String other(int n) {
    return "Other "+fullDesc+" description..."+n;
 }
```

Practice Problems

- Update your Quadrant enumeration to have two private fields for the x and y Signs. (Bonus complexity: use your Sign enumeration for these!)
- Add constructor calls to use these fields for your Quadrant values.
- write a toString method for your Quadrant enumeration.
- write two methods: getXSign and getYSign. They work on a Quadrant value (no parameters), returning the Sign value for an xx/y value in this quadrant.

Exceptions

Exceptions: Specialized Control Flow

Exceptions: The Idea

Exceptions as a Class Hierarchy

Creating Exception values

Handling Exceptions

Exceptions: The Idea

Sometimes, exceptional events occur during execution of our program. For example:

- array index is out of bounds
- we divided by zero
- we tried to use null as a reference to an object
- we tried to open a non-existing file

Normal control flow is aborted, in search of a way to handle the exceptional event.

- We keep escaping code blocks until one is found.
- If we escape all the way out of main, the program crashes.

Exception Types: A Class Hierarchy

- A **small portion** of the huge class hierarchy of Exceptions already defined in Java.
- You've probably already seen a few of these.

- **java.lang.Throwable** *(implicitly inherits from Object).*
  - **Exception**
    - **RuntimeException** *for recoverable events.*
      - **NullPointerException** *using null like an object*
      - **ClassCastException** *cast to class-type that wasn't possible*
      - **IndexOutOfBoundsException**
        - **ArrayIndexOutOfBoundsException**
        - **StringIndexOutOfBoundsException**
      - **ArithmeticException** *bad arithmetic, like "divide by zero"*
    - **IOException**
      - **FileNotFoundException** *attempted to open non-existing file*
      - **EOFException** *end of file reached (no more content)*
  - **Error** *unrecoverable events: e.g., out of memory*

Getting/Creating Exception Values

from methods: Some Exception values are created inside methods you call. For example:

- `FileNotFoundException` thrown by FileInputStream constructor when the file is not found.

basic Java expression usage: Some Exception values are created through incorrect value usage. Examples:

- dividing by zero will cause an `ArithmeticException`
- using an out-of-bounds index will cause an `ArrayIndexOutOfBoundsException`

creating your own: You can create your own:

- call the constructor of an Exception class. You'll also need to 'throw' it:

```
ArithmeticException ae = new ArithmeticException("evens only!");
throw ae;
```

Practice Problems

What are some situations where you can imagine Java creating/throwing exceptions?

When would you want to manually generate an exception type that Java already has?

When would you want to create your own types of Exceptions and throw them?

How many exceptions can escape one block of code?

Handling Exceptions

We have two options:

Catch It: wrap the offending code in a `try-catch` block that catches the specific type of exception.

Propagate It: allow the exception to occur, crashing its way through your program until it is caught elsewhere.

- you might have to explicitly list what exceptions are propagated (any exception that is not a `RuntimeException` or `Error`).

No matter what, the occurring exception immediately starts 'crashing' your program by prematurely leaving each code block and method call, until it is caught by a catch-block (or the entire program is crashed).

Catching Exceptions: `try-catch` Blocks

Wrap the suspicious code in a **`try`**-block.

Provide a way to handle the occurring exception with a **`catch`**-block. This must include the type of Exception being caught.

→ if the exception occurs in the try-block, the catch block runs.

```
try {
   int infinity = 5 / 0 ;
   System.out.println("I'm never printed. " + infinity);
}
catch (ArithmeticException e) {
   System.out.println("saw arith. error: " + e);
}
```

Practice Problems

Write code using a `try-catch` block that successfully gets an integer from the user, using a `Scanner`.

Write a method containing a `try-catch` block that accepts a String argument and tries to parse an `int` out of it to return. Return -1 if the parsing fails. (What Exception to catch?)

Write a method accepting a Square parameter that tries to print it to the terminal. Using a try-catch block, if the Square value is null, just print "<no Square>" instead.

Multiple `catch`-Blocks

We can add multiple `catch`-blocks. The **first** block that can handle the exception that actually occurred is the **only** `catch`-block to run.

```
public String makeAString (int[] xs, int starterIndex) {
    String retval = "";
    try {
        int myIndex = 50 / starterIndex;  //might divide by zero
        int rval = xs [ myIndex ] ;       //index maybe out of bounds
        retval = "result: " + retval ;
    }
    catch (ArithmeticException e) {retval = "div. by zero."; }
    catch (ArrayIndexOutOfBoundsException e) {retval = "bad index."; }
    catch (Exception e) { retval = "other error..."; }
    return retval;
}
```

inputs:	exception:
{2,4,6}, 0	ArithmeticException (/ by zero)
{2,4,6}, 5	ArrayIndexOutOfBoundsException
null, 5	NullPointerException → this is also an Exception

finally Blocks

A finally-block always runs, whether the try-block is successful, or an exception is caught, or an exception is propagated.

```
int ans = -1;
int[] xs = {3,5,0,6,4};
try {
    int temp = sc.nextInt();
    if (temp<5) {
        ans = 25 / xs[temp];
    }
    System.out.println("success!");
    return ans;
}
catch (ArithmeticException e){rval="div. by zero.";}
finally {  // whether success, div by zero, or index error.
    System.out.println("I always run. Always.");
}
```

Thoughts

On the previous slide, what is printed when the scanner input's next `int` is each of these:

-2 0 2 4 6

Is it possible for a finally block to not execute?

can we have return statements in a:

- → `try` block?
- → `catch` block?
- → `finally` block?

Reminder: Exceptions are Abrupt!

When an exception is thrown, we *immediately* cease executing the current block of code. The following 'exits' occur repeatedly (in this order), until the exception is handled:

- We don't finish any expression simplifications of the current statement
- We skip to the end of the `try`-block (if we're in one)
- We skip any non-matching `catch` blocks (if there are any)
- We exit the method call with our exception value
- We managed to escape the main method, and crash the entire program. (traceback printed to System.err)

Exceptions and Side-Effects

What effects happen when an exception occurs?

All side effects prior to thrown exception still occurred (e.g., assignments & printing).

Statements directly after the offending line are not run: the exception is propagating instead of the intended control flow.

If the exception is from an expression within a statement, the statement isn't even completed!
→ `y=5/0;` does not actually assign a value to `y`.

Practice Problems

What is printed by the following?

```
int x = 1, y=500;

try {
        x=6;
        y = 50 / 0;
        x++;
}
catch (Exception e) {
        y++;
}
finally {
        y += 30;
}
System.out.print("x="+x+", y="+y);
```

x=6, y=531

```
int x = 0;
int xs[] = {3,5,7};
try {
        x =  50 / 0;
        xs[50] = 3;
}
catch (ArithmeticException e) {
        x += 20;
}
catch(ArrayIndexOutOfBoundsException e) {
        x += 300;
}
catch (Exception e) {
        x += 4000;
}
finally { x+= 50000;}
System.out.print("x="+x);
```

x=50020

Propagating Exceptions

We can choose not to handle an exception, propagating it.

If an exception is unhandled (propagated), then when it occurs, our method might abnormally return with a crashing exception instead of its intended return value, and instead of completing the intended control flow and calculations.

There are two kinds of Exceptions:
Checked Exceptions and **Unchecked Exceptions**.

Kinds of Exceptions

Checked Exceptions: we must annotate the method signature admitting that our code could cause an exception. You can list many unhandled exceptions this way for a single method.

```
public int lazyBum (…)throws FileNotFoundException{..}

public int foo () throws EOFException, IOException, NullPointerException{..}
```

Unchecked Exceptions can also be unhandled, but don't require the throws annotation: no try-catch block used around the suspicious code, and no further annotations. Only **`Error`**, **`RuntimeException`** and their child classes are unchecked. (You may still list them in the throws clause, as we did above for `NullPointerException`).

- Error examples: failing disk, not enough memory; program can't solve these.
- Runtime Exceptions: good indication of program bugs, not behaviors we expect in a 'correct' program; their causes should be fixed.

Creating Your Own Exception Classes

- Exception definitions are classes. Extend these classes to make your own specialized exceptions.
- use fields, write constructors to pass info around.

```
public class MyException extends Exception {
        public int num;
        public String msg;
        public MyException (int num, String msg) {
                this.num = num;
                this.msg = msg;
        }
}
```

Using Your Own Exceptions

create values by calling the constructor.

```
MyException myExc = new MyException(5,"yo");
```

begin exception propagation with a **throw** statement:

```
throw myExc;
```

or create and throw, all at once:

```
throw new MyException(6,"hiya!");
```

Practice Problems

Create your own Exception classes, named `OutOfFoodException` and `OutOfCheeseException`.

- What should they extend?

Add fields and constructors to each.

Create and throw values of each; write catch blocks that successfully catch each one.

- can you write catch blocks that catch them without explicitly writing "catch (OutOfFoodException e)" or "catch (OutOfCheeseException e)" ?