Object Oriented Programming with Java

Exceptions

Exceptions: The Idea

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

- array index is out of bounds
- we divided by zero
- we tried to open a non-existing file
- many others…

Normal sequential 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 perhaps already seen a few of these.

java.lang.Throwable	*(implicitly inherits from Object).*
Exception	
RuntimeException	*for recoverable events.*
NullPointerException	*tried 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*

Creation of Exception Values

- **automatic creation**

  Some Exception values are automatically created by an error caused during runtime. Examples:

  - **dividing by zero** will cause an `ArithmeticException`
  - using a **wrong index** will cause an `ArrayIndexOutOfBoundsException`
  - opening a **file that doesn't exist** will cause a `FileNotFoundException`

```
File f = new File("this_file_does_not_exist.txt");
```

- **intentional creation**

  You can manually create an Exception value by instantiating an object of an Exception class and then 'throw' it:

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

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("arithmetic error: " + e);
}
```

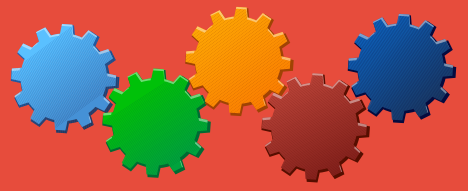
Handling Exceptions

We have two options:

- **Catch It:** wrap the offending code in a **`try-catch`** block that catches the specific type of exception.
- **Defer It:** allow the exception to occur, propagating (*crashing*) its way through your program until it is caught elsewhere.
  - you might have to explicitly list what exceptions are deferred

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*).

Practice Problems



- Write code using a `try-catch` block that successfully gets an integer from the user, using a `Scanner`. Then, add logic to make it a "pin validation" program (e.g. for a smartphone).

- 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;
        int rval = xs[myIndex];
        retval = "result: " + retval;
    }
    catch (ArithmeticException e) {retval = "div. by zero."; }
    catch (ArrayIndexOutOfBoundsException e) {retval = "bad index."; }
    catch (Exception e) { retval = "other error..."; }
    return retval;
}
```

Input	Output ?
`{2,4,6}, 0`	`ArithmeticException (division by zero)`
`{2,4,6}, 5`	`ArrayIndexOutOfBoundsException`
`null, 5`	`NullPointerException → this is also an Exception`

Multi-catch Block

We can also group more than one exceptions in a single catch-block
(*since Java 7*)

```
String getVal() {
    try {
        return array[scanner.nextInt()/scanner.nextFloat()];
    }
    catch (ArithmeticException e | ArrayIndexOutOfBoundsException e) {
        return "division by zero or bad index";
    }
    catch (Exception e) {
        return "other error...";
    }
}
```

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 {
    System.out.println("I always run. Always.");
}
```

Note: The `finally` block will execute even if there is a return statement inside `try` or `catch`. It will only be skipped with **`System.exit()`**

Practice Problems

- 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:

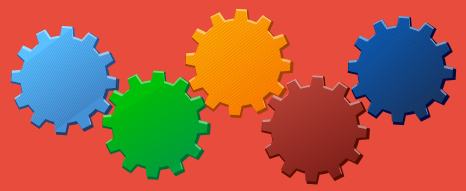
1. We skip to the end of the `try`-block (if we're in one)
2. We skip any non-matching `catch` blocks (if there are any)
3. We exit the method call with our exception value
4. We managed to escape the main method, and crash the entire program. (traceback printed to `System.err`)

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 and let it **propagate** back to the calling environment so that someone else catches it.

```
void method1() {
    try {
        method2();
    }
    catch (ArrayIndexOutOfBoundsException e) {
        x = -1;
    }
}

void method2() {
    try {
        x = array[sc.nextInt()] / sc.nextFloat();
    }
    catch (ArithmeticException e) {
        x = 0.0;
    }
}
```

Any exception that can still escape in this case?

Checked vs Unchecked Exceptions

- Exceptions in Java are grouped in two categories:
  - **Checked Exceptions** and **Unchecked Exceptions**

- **Checked** exceptions are meant for errors that are outside the control of the program (e.g. open a network connection that doesn't respond)

- **Unchecked** exceptions are meant for errors in the logic of the program (e.g. divide a number by zero)

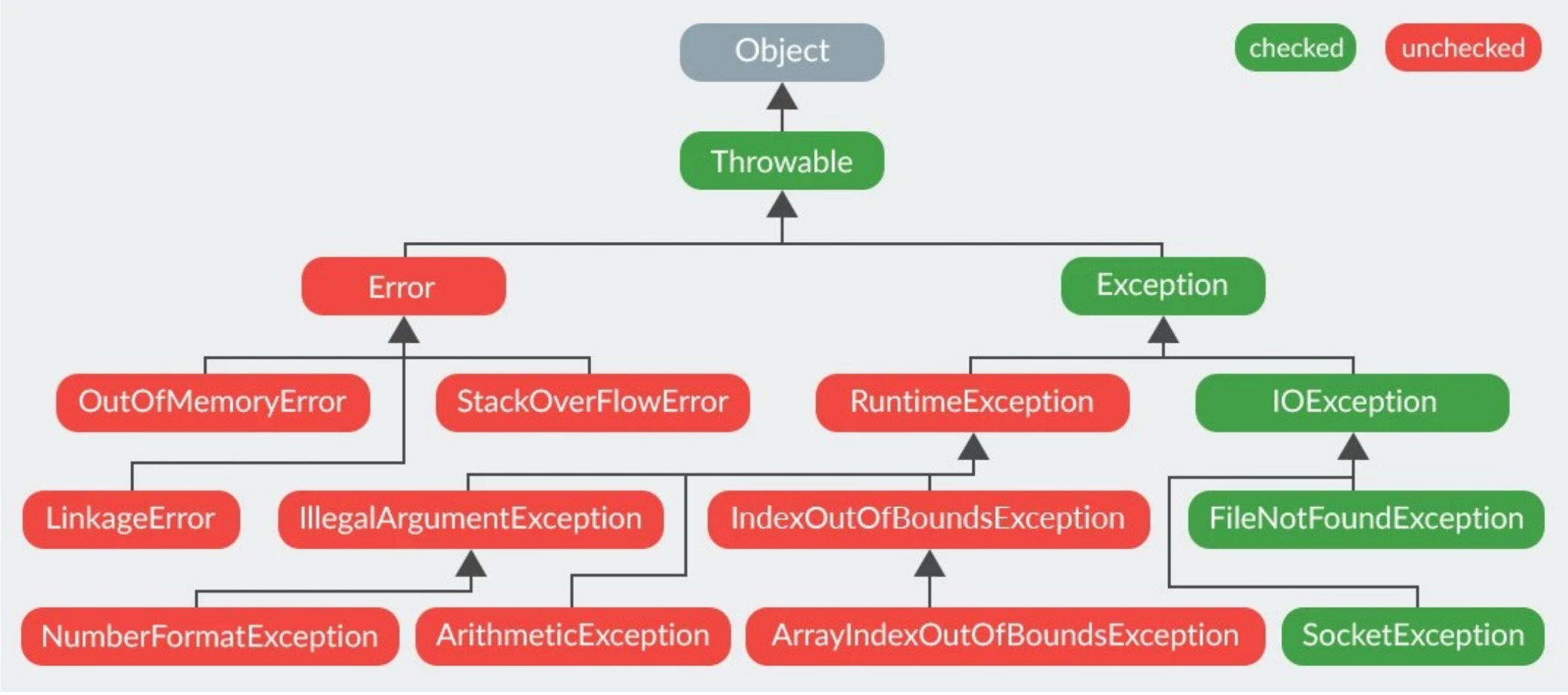
Checked vs Unchecked Exceptions

- **Checked** exceptions **must be checked** at compile time. If some code within a method throws a checked exception, then the method must do one of the following:
  - handle the exception with a `try-catch` block
  - or declare that the method potentially `throws` an exception (**note** that keyword `throws` is different from keyword `throw` that is used for throwing an exception).

```
public void someMethod() throws FileNotFoundException
{
    // code goes here
}
```

- **Unchecked** exceptions **are not required** to be checked at compiled time, but it's a good practice to still do check them.
  - All `Error` classes and the whole hierarchy under `RuntimeException`

Checked vs Unchecked Exceptions



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;
    }
}
```

Should we make our exceptions **checked or unchecked**?

- If a client can reasonably be expected to recover from an exception, make it *checked*
- If a client cannot do anything to recover from the exception, make it *unchecked*

Using Your Own Exceptions

- create values by calling the constructor.

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

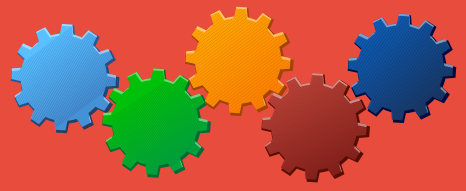
- begin exception propagation with a **throw** statement:

```
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)`