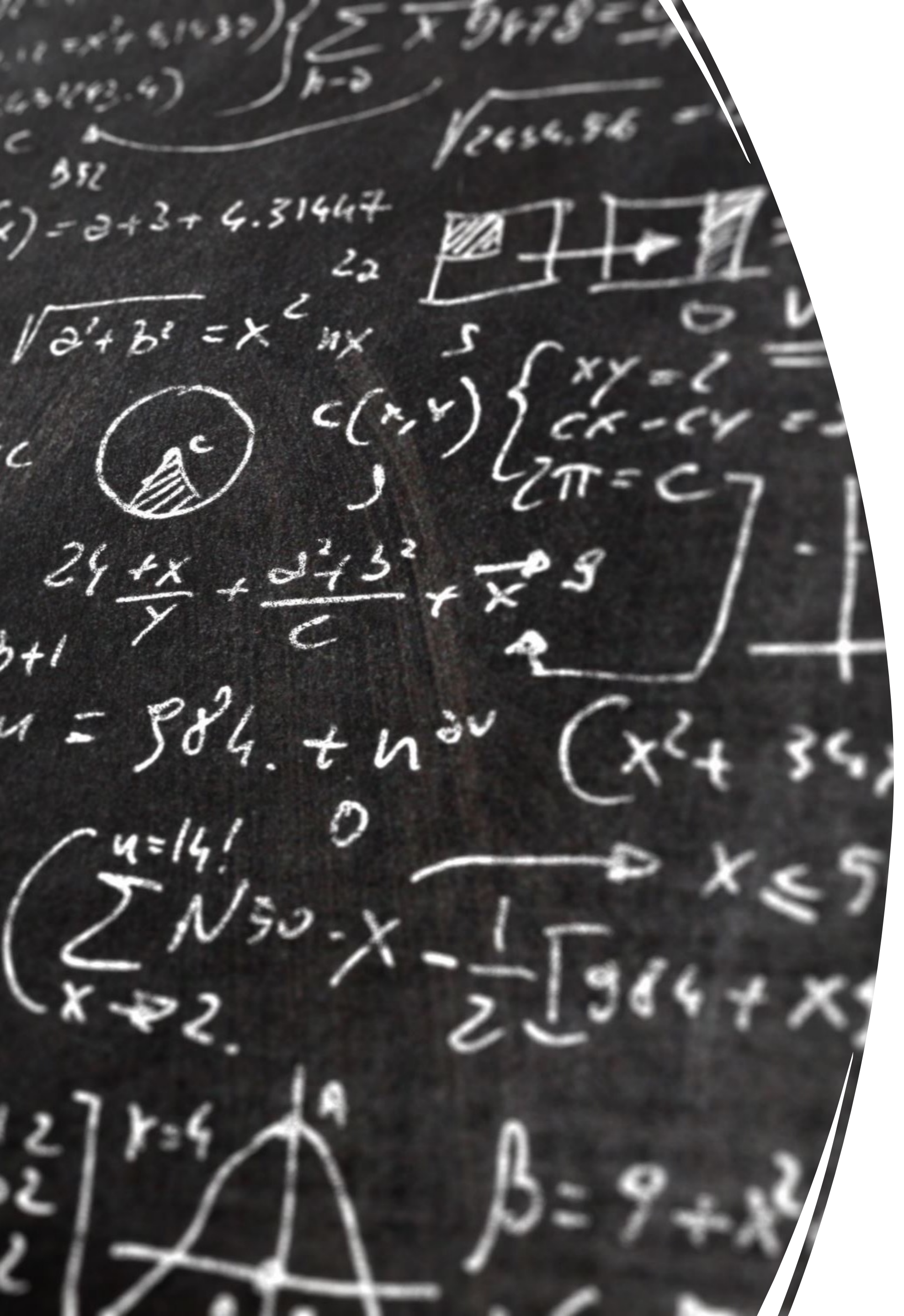


CS 113 – Computer Science I

Lecture 23 – Exceptions

Adam Poliak

12/06/2022

Announcements

- Assignment 11
  - Due Thursday 12/08
  - Optional/extra credit

- Lab: additional office hours

Exercise

Write a program, Cake.java, that implements a Cake class that stores a cake name and cost. In main(), read in a CSV file of cakes into an ArrayList and sort them from least expensive to most expensive.

```
$ java-introcs Cake cakes.txt
Red velvet cake: $2.0
Chocolate cake: $3.5
Strawberry cake: $4.5
Cheesecake: $6.99
```

Sedgewick and Wayne, page 465

Exceptions

An **exception** is a disruptive event that occurs while a program is running

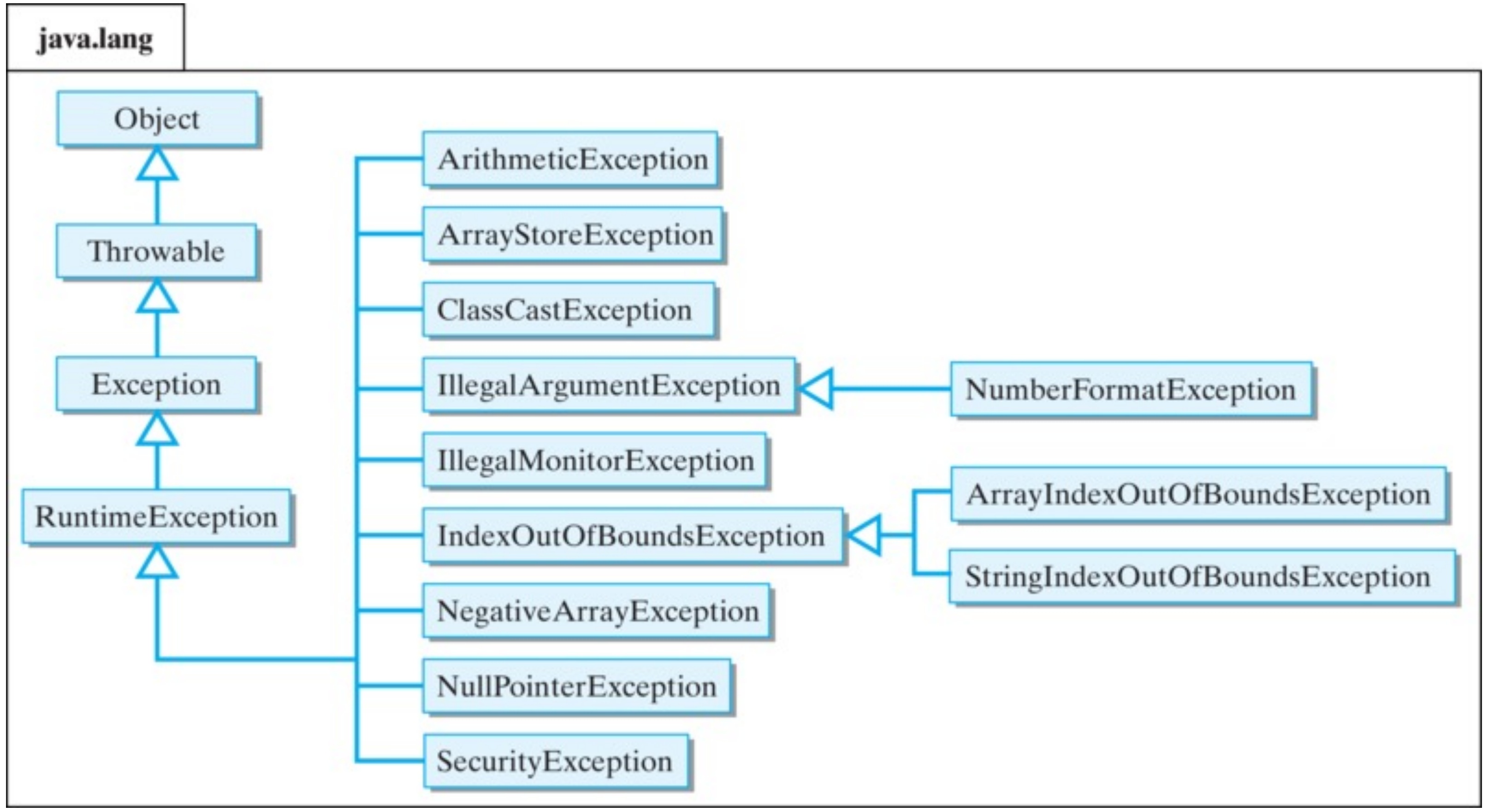
typically indicates a *runtime error*

Examples: IndexOutOfBoundsException, NumberFormatException

When an error occurs, we **throw** the exception. Any function that is currently on the stack can **catch** the exception.

- Functions that do not catch the exception are aborted
- If no one catches the exception, the program terminates and prints the exception to the console

Exceptions are objects



Throwing an exception

```
public static void bar() {
  throw new RuntimeException("An error happened in bar()");
}
```

Catching an exception

```
    try {
      bar();
    }
    catch (RuntimeException e) {
      System.out.println("An exception occured: "+e.getMessage());
      e.printStackTrace();
    }
```

Draw the stack diagram

```
public static void bar() {
  throw new RuntimeException(“ERROR");
}

public static void foo() {
  try {
    bar();
  }
  catch (RuntimeException e) {
    System.out.println(“Exception: "+e.getMessage());
    e.printStackTrace();
  }
  System.out.println("Hello!");
}

public static void main(String[] args) {
  foo();
}
```

Exercise: Write a program that catches an ArrayOutOfBoundsError

Exceptions: best practices

- A production-level application should never throw and uncaught exception
  - e.g. the user should never encounter an exception.
  - thrown exceptions are bugs

- Throwing an exception is meant to help the developer
  - Serious mistakes that will derail further execution of the program
  - Errors related to undefined behaviors typically throw exceptions
    - divide by zero
    - adding vectors with mis-matches sizes
    - out of array bounds

Exceptions: best practices

```
class CheckInteger {
  public static void main(String[] args) {

    int value = 0;
    boolean valid = false;
    while (!valid) {
      System.out.print("Enter an integer: ");
      String input = System.console().readLine();
      try {
        value = Integer.parseInt(input);
        valid = true;
      }
      catch (RuntimeException e) {
        System.out.println("Sorry, this value is invalid");
      }
    }

    System.out.println("You entered "+value);
  }
}
```

NO

Exceptions are slow and should not be used for routine error checking

- For example, checking whether a user input an integer