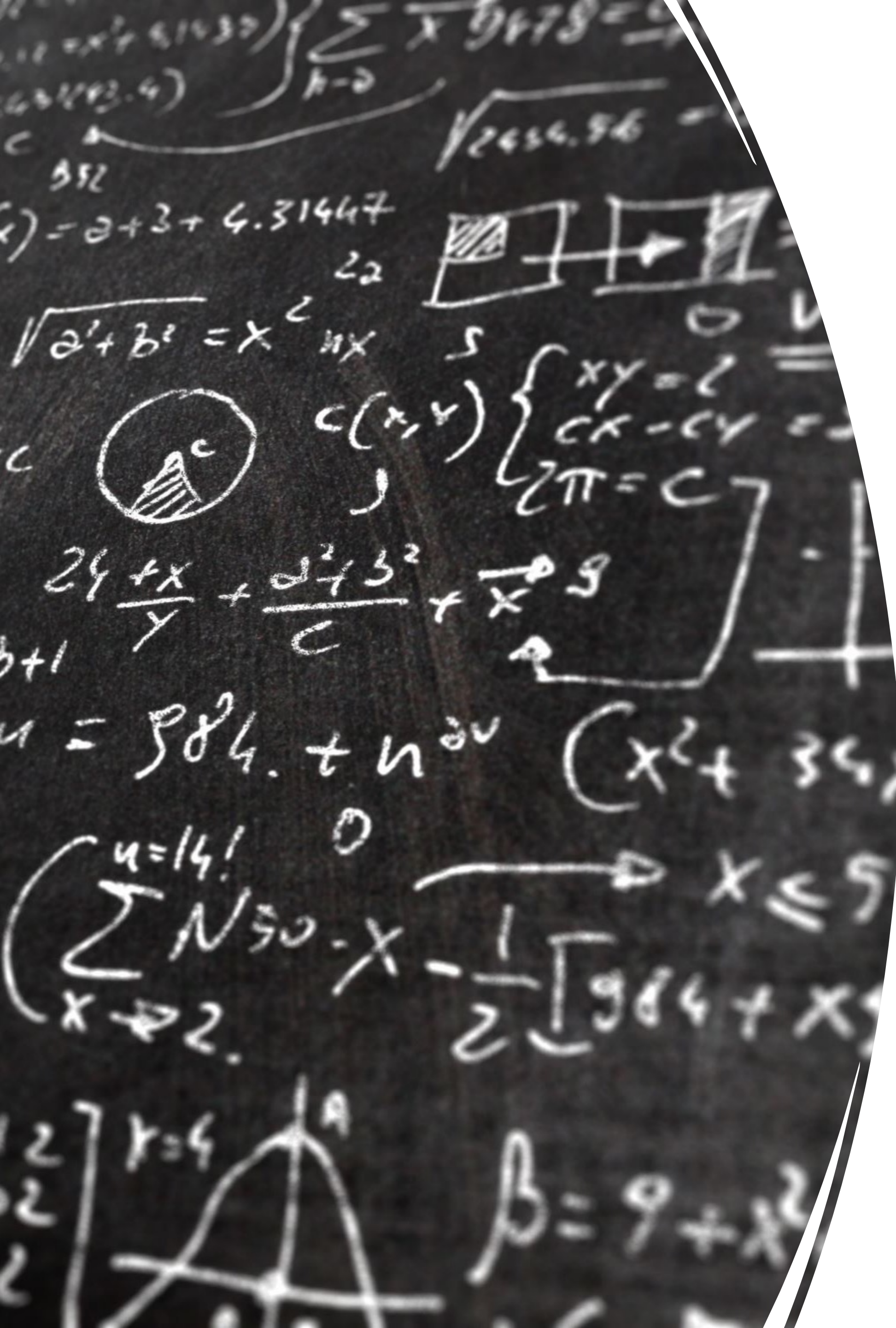


CS 113 – Computer Science I

Lecture 21 – Data Structures

Adam Poliak

11/29/2022

Announcements

- Assignment 10
  - Due Thursday 12/01
- Assignment 11
  - Due Thursday 12/08
  - Optional/extra credit
- Code jam this week in lab

Data structures

Containers for data

What data structures did we use so far this term?

How we organize data in our programs matters a lot

- Effects performance
- Can make a feature easier or harder to implement

Arrays

Properties of arrays:

- Built-in type in Java
- all items must be the same time
- Access/set items by index
- Ordering of data
- Multiple dimensions
- the size of the array is fixed
  - What happens when we want to add more elements to an array?
    - Crreate new array with more space
    - Copy elements from smaller array to new bigger array
    - Runtime
      - O(n) – linear

ArrayList

Properties of ArrayList:

- Built-in type in Java
- all items must be the same time (can overcome but beyond scope of CS113)
- Access/set items by index
- Ordering of data
- Multiple dimensions
- the size of the array is not fixed

ArrayList

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

```
public static void main(String[] args) {
     ArrayList<Integer> list = new ArrayList<Integer>();
     list.add(10);
     list.add(-7);
     list.add(3);

     for (int i = 0; i < list.size(); i++) {
          int v = list.get(i);
     }

     boolean test = list.contains(-7);
     list.remove(1);

     for (int i = 0; i < list.size(); i++) {
          int v = list.get(i);
     }
     list.clear();
}
```

ArrayList - initialize

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

```
public static void main(String[] args) {
     ArrayList<Integer> list = new ArrayList<Integer>();
     list.add(10);
     list.add(-7);
     list.add(3);

     for (int i = 0; i < list.size(); i++) {
          int v = list.get(i);
     }

     boolean test = list.contains(-7);
     list.remove(1);

     for (int i = 0; i < list.size(); i++) {
          int v = list.get(i);
     }
     list.clear();
}
```

ArrayList – adding elements

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

```
public static void main(String[] args) {
     ArrayList<Integer> list = new ArrayList<Integer>();
     list.add(10);
     list.add(-7);
     list.add(3);

     for (int i = 0; i < list.size(); i++) {
          int v = list.get(i);
     }

     boolean test = list.contains(-7);
     list.remove(1);

     for (int i = 0; i < list.size(); i++) {
          int v = list.get(i);
     }
     list.clear();
}
```

ArrayList – accessing elements

```
public static void main(String[] args) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(10);
    list.add(-7);
    list.add(3);

    for (int i = 0; i < list.size(); i++) {
        int v = list.get(i);
    }

    boolean test = list.contains(-7);
    list.remove(1);

    for (int i = 0; i < list.size(); i++) {
        int v = list.get(i);
    }
    list.clear();
}
```

ArrayList – other helper methhods

```
public static void main(String[] args) {
     ArrayList<Integer> list = new ArrayList<Integer>();
     list.add(10);
     list.add(-7);
     list.add(3);

     for (int i = 0; i < list.size(); i++) {
          int v = list.get(i);
     }

     boolean test = list.contains(-7);
     list.remove(1);

     for (int i = 0; i < list.size(); i++) {
          int v = list.get(i);
     }
     list.clear();
 }
```

Visualizing ArrayList – Stack Frame

```
public static void main(String[] args) {
     ArrayList<Integer> list = new ArrayList<Integer>();
     list.add(10);
     list.add(-7);
     list.add(3);

     for (int i = 0; i < list.size(); i++) {
         int v = list.get(i);
     }

     boolean test = list.contains(-7);
     list.remove(1);

     for (int i = 0; i < list.size(); i++) {
         int v = list.get(i);
     }
     list.clear();
}
```

Exercise

Write a program, Average.java, that stores values in a list until the user presses RETURN (e.g. empty string). Then print all values that are less than the average.

```
$ java Average
Enter an integer, or 'RETURN' to quit: 3
Enter an integer, or 'RETURN' to quit: 34
Enter an integer, or 'RETURN' to quit: 10
Enter an integer, or 'RETURN' to quit: 90
Enter an integer, or 'RETURN' to quit: 102
Enter an integer, or 'RETURN' to quit: 22
Enter an integer, or 'RETURN' to quit: 75
Enter an integer, or 'RETURN' to quit:
The average value is 48.0
3 is less than the average.
34 is less than the average.
10 is less than the average.
22 is less than the average.
```

ArrayList

- Convenient when we don’t know the size we need at the start
- Best for storing sequences/list of data
- When we run out of space, the array list resizes itself
- Adding elements to the end is generally fast (so long as we don’t need to resize)
- Removing elements or inserting in the middle can be slow (need to shift elements)

Hashmap

Stores <key, value> pairs

Examples: associate a name to age

Examples: associate a studentId to a grade

Fast lookup, add, and remove by key

Does not preserve the ordering of data

Keys should be unique

Hashmap

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

```
public static void main(String[] args) {
    HashMap<String, String> map = new HashMap<String, String>();
    map.put("dog", "woof");
    map.put("cow", "moo");
    map.put("cat", "meow");
    map.put("bird", "chirp");
    System.out.println(map.get("dog"));

    for (String key : map.keySet()) {
        System.out.printf("What does the %s say? %s\n", key, map.get(key));
    }

    bool test = map.containsKey("turkey");
    System.out.println(test);

    map.remove("cat");
}
```

Hashmap – adding to hashmap

```
public static void main(String[] args) {
    HashMap<String, String> map = new HashMap<String, String>();
    map.put("dog", "woof");
    map.put("cow", "moo");
    map.put("cat", "meow");
    map.put("bird", "chirp");
    System.out.println(map.get("dog"));

    for (String key : map.keySet()) {
        System.out.printf("What does the %s say? %s\n", key, map.get(key));
    }

    bool test = map.containsKey("turkey");
    System.out.println(test);

    map.remove("cat");
}
```

Hashmap – accessing from hashmap

```
public static void main(String[] args) {
    HashMap<String, String> map = new HashMap<String, String>();
    map.put("dog", "woof");
    map.put("cow", "moo");
    map.put("cat", "meow");
    map.put("bird", "chirp");
    System.out.println(map.get("dog"));

    for (String key : map.keySet()) {
        System.out.printf("What does the %s say? %s\n", key, map.get(key));
    }

    bool test = map.containsKey("turkey");
    System.out.println(test);

    map.remove("cat");
}
```

Hashmap – iterating through hashmap

```
public static void main(String[] args) {
    HashMap<String, String> map = new HashMap<String, String>();
    map.put("dog", "woof");
    map.put("cow", "moo");
    map.put("cat", "meow");
    map.put("bird", "chirp");
    System.out.println(map.get("dog"));

    for (String key : map.keySet()) {
        System.out.printf("What does the %s say? %s\n", key, map.get(key));
    }

    bool test = map.containsKey("turkey");
    System.out.println(test);

    map.remove("cat");
}
```

Hashmap – searching in a hashmap

```
public static void main(String[] args) {
    HashMap<String, String> map = new HashMap<String, String>();
    map.put("dog", "woof");
    map.put("cow", "moo");
    map.put("cat", "meow");
    map.put("bird", "chirp");
    System.out.println(map.get("dog"));

    for (String key : map.keySet()) {
        System.out.printf("What does the %s say? %s\n", key, map.get(key));
    }

    boolean test = map.containsKey("turkey");
    System.out.println(test);

    map.remove("cat");
}
```

Visualizing Hashmaps

```
public static void main(String[] args) {
    HashMap<String, String> map = new HashMap<String, String>();
    map.put("dog", "woof");
    map.put("cow", "moo");
    map.put("cat", "meow");
    map.put("bird", "chirp");
    System.out.println(map.get("dog"));

    for (String key : map.keySet()) {
        System.out.printf("What does the %s say? %s\n", key, map.get(key));
    }

    bool test = map.containsKey("turkey");
    System.out.println(test);

    map.remove("cat");
}
```

Exercise

Write a program, CountLetters.java, that counts the number of times each character appears in a given string.

```
$ java LetterCount
Please enter a word: lol
l: 2
o: 1

$ java LetterCount
Please enter a word: abba
a: 2
b: 2
```

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
```

Exceptions

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

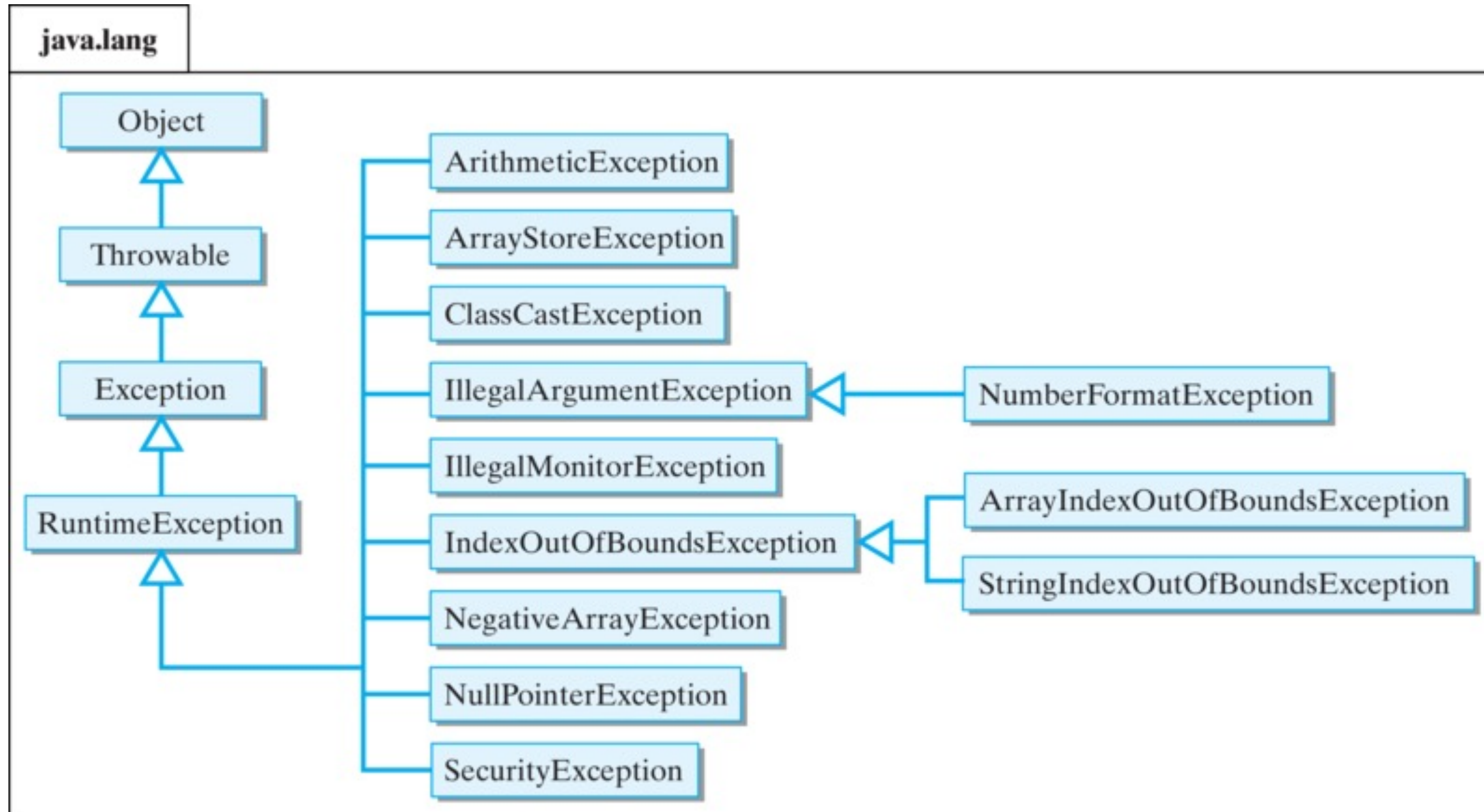
typically indicates a *runtime error*

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

```java
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

Exceptions: Trace this program

```
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(“ERROR");
  }
}

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