CS 113 – Computer Science I

Lecture 15 – Arrays, Classes & Objects

Tuesday 10/31/2023
Announcements

HW 06 – Due Monday 11/06
• Loops
Agenda

• Arrays
• Mutability
• Objects & Classes
Bank example

Keep track of account balances

Use an array:
   Each index represents another account
   The value represents the account’s balance

Determine how many accounts we can hold:
   Create a new array of fixed size
Bank

How can we find out how much money the bank is holding at once?

How can we find out which account is currently overdraft?

What other questions might the bank want to know?
Bank example

Over time our bank becomes successful, lots of new clients

No more space for new customers

Implementation issue: running out of space in our array

Solution: build a bigger bank!
Building a bigger bank
Copying arrays

<table>
<thead>
<tr>
<th>Old bank</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
<td>6.0</td>
<td>7.0</td>
<td>-2.5</td>
</tr>
</tbody>
</table>
Copying arrays – build the new bank/array

Old bank

| 3.0 | 6.0 | 7.0 | -2.5 |

new bank
Copying arrays – copy over values/customers

Old bank

3.0  6.0  7.0  -2.5

new bank
Copying arrays – copy over values/customers

Old bank

3.0  6.0  7.0  -2.5

new bank
Copying arrays – copy over values/customers

Old bank

3.0  6.0  7.0  -2.5

new bank

3.0

Copying arrays – copy over values/customers

Old bank

| 3.0 | 6.0 | 7.0 | -2.5 |

new bank

3.0
Copying arrays – copy over values/customers

Old bank

| 3.0 | 6.0 | 7.0 | -2.5 |

new bank

| 3.0 | 6.0 |   |   |
Copying arrays – copy over values/customers

Old bank

3.0  6.0  7.0  -2.5

new bank

3.0  6.0
Copying arrays – copy over values/customers

Old bank

| 3.0 | 6.0 | 7.0 | -2.5 |

new bank

| 3.0 | 6.0 | 7.0 |   |   |
Copying arrays – copy over values/customers

Old bank

| 3.0 | 6.0 | 7.0 | -2.5 |

new bank

| 3.0 | 6.0 | 7.0 |     |     |
Copying arrays – copy over values/customers

Old bank

| 3.0 | 6.0 | 7.0 | -2.5 |

new bank

| 3.0 | 6.0 | 7.0 | -2.5 |
Algorithm

When we run out of space in an array
• Create a new array (that’s a bit bigger)
• Copy over all elements from the older array to the new array

How many steps do we take in this algorithm?
• Creating a new array – 1 step
• Copying $n$ elements from the old array to the new array – $n$ steps
How big should the new array be?

Previous size plus 1
• Pro: not making too much space
• Con: might have to create new arrays a lot of times

As big as possible
• Pro: rarely have to create a new array
• Con: wasted space

Typical solution – previous size x 2
Agenda

• Arrays
• **Mutability**
• Objects & Classes
Mutable vs Immutable

Mutable:
   Values can change

Immutable:
   Values cannot change

Strings and Integers are immutable
Agenda

• Arrays
• Mutability
• Objects & Classes
Data types revisited

What are some examples of built-in types in Java?

What is a data type?
# Examples

<table>
<thead>
<tr>
<th>Type</th>
<th>Valid values</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Examples

<table>
<thead>
<tr>
<th>Type</th>
<th>Valid values</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples

<table>
<thead>
<tr>
<th>Type</th>
<th>Valid values</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>1, 10, 999</td>
<td>%, +, -, / ...</td>
</tr>
</tbody>
</table>
# Examples

<table>
<thead>
<tr>
<th>Type</th>
<th>Valid values</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>1, 10, 999</td>
<td>%, +, -, / ...</td>
</tr>
<tr>
<td>boolean</td>
<td>true, false</td>
<td>==, &amp;&amp;,</td>
</tr>
<tr>
<td>String</td>
<td>Anything between &quot;&quot;</td>
<td>.compareTo(), .charAt(), concatenation, ...</td>
</tr>
</tbody>
</table>
Class

A blueprint for a custom data type

A template for how data/information is stored

Contains a set of methods for how to interact/operate on the stored data
Classes and objects

An object is an *instance* of a class

An object is to a class as a

- cat is to an animal
- tulip is to an flower
- cookie it to a snack
- Socrates is to a human
Classes and objects

A **class** defines the characteristics of a type (data and methods)

An **object** is a particular example of a class

Java is a strict object-oriented programming language, meaning all code must be inside a class!
Creating objects

Declare variables in the same way!

Create using `new`
Using objects

The methods you are allowed to call on an object is called an **API**
Recall: API = Application Programming Interface

Example: The *String API* has over 60 methods!

Objects can have either *static* or *instance* methods

- static methods use syntax `<ClassName>.<methodName>`
- instance methods use syntax `<object>.<methodName>`
Example: String API

<table>
<thead>
<tr>
<th>boolean</th>
<th>endsWith(String suffix)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests if this string ends with the specified suffix.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>boolean</th>
<th>equals(Object anObject)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compares this string to the specified object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>boolean</th>
<th>equalsIgnoreCase(String anotherString)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compares this String to another String, ignoring case considerations.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>static String</th>
<th>format(Locale l, String format, Object... args)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns a formatted string using the specified locale, format string, and arguments.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>static String</th>
<th>format(String format, Object... args)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns a formatted string using the specified format string and arguments.</td>
</tr>
</tbody>
</table>
Using objects: some special methods

The **constructor method** is called when you do a `new`

**accessors (aka getters)**
return the values of instance variables

**mutators (aka setters)**
set the values of instance variables

**toString()**
returns a string representation of an object
Defining classes

By defining our own classes, we can create our own data types

A class definition contains

- the data contained by the new type (instance variables)

- the operations supported by the new type (instance methods)
OOP Example & Design: Bank
Defining the Bank class

```java
public class Bank {
    private int size;
    private String name;
    private String[] clients;
    private double[] accounts;

    public Bank(String bankName, int numClients) {
        name = bankName;
        size = numClients;
        clients = new String[size];
        accounts = new double[size];
    }

    public String getName() {
        return name;
    }
}
```
Object-oriented programming (OOP)

Method for designing programs in terms of objects

Recall: Top-down design

- the “nouns” in your feature list correspond to classes/data
- the “verbs” correspond to methods
public static void main(String args[]) {
    Bank boa = new Bank("Bank of America", 10);
    System.out.println("Bank: "+boa.getName());
}
Objects: Stack diagrams revisited

```java
public static void main(String args[])
{
    Bank boa = new Bank("Bank of America", 10); //call constructor
    System.out.println("Bank: "+boa.getName());
}
```

Exercise: draw a stack diagram for this program
Exercise: Define a class BankAccount

BankAccount should have the following data:

• Name
• Amount

BankAccount should have the following operations:

• currentBalance() // returns current amount in the bank account
• withdraw(float amt) // withdraw the given amount from the account
• deposit(float amt) // deposit the given amount to the account