Lecture 13
Arrays

Outline
• Purpose
• Array Syntax
• ArrayLists
• Multi-Dimensional Arrays

Why Use Arrays? (1/2)
• So far, we've only studied variables that hold references to single objects
• What about holding lots of data? Many programs need to keep track of hundreds/thousands of data instances
  o can't always assign unique name to each data item (Brown students have names, data from an experiment don't)
• Want to hold arbitrary number of objects with single reference — represents a collection of elements
  o allows for simple communication to multiple elements
  o but still need to have unique identification of any element
• Arrays are the simplest data structure or collection — we'll also cover lists, queues, and stacks
Why Use Arrays? (2/2)

- Arrays allow instances of specific type to be “packaged” together and accessed as group

- What if there are 13 instances of `District`?
  - store all `Districts` in array for easy access (to make sure we have the right number of tributes from each district)

- Arrays are ordered - helpful when wanting to store or access instances in particular order, e.g., alphabetically

Your lovely TAs

- We want access to all 59 UTAs (and 5 HTAs)!
  - Adam, Asia, …, Xiaoyue

- Could use instance variables:
  ```java
  public class CS15TAs {
      private TA adam;
      private TA asia;
      // other TAs elided
      private TA xiaoyue;
  }
  ```

- Can’t access 64 instance variables very easily
  - what if we wanted to access CS15 TAs from fall 2022, 2021, 2020, …

Arrays (1/4)

- Arrays store specified, constant number of data elements of same type – our first homogeneous collection
  - each element must be same type or subclass of same type (polymorphism)

- Arrays are special in Java
  - special syntax to access array elements:
    ```java
    studentArray[index]
    ```
  - the `index` of array is always of type `int`
  - neither base type nor class, but Java `construct`
    ```java
    // use new to initialize an array (even though it's not a class)
    ```
  - special syntax, does not invoke constructor like for a class
Arrays (2/4)
- Arrays only hold elements of specified type
  - when declaring arrays, state type of object it stores:
    - base type
    - class
    - sub-arrays (for multi-dimensional arrays – soon)
    - or for max polymorphic flexibility, interface or superclass
  - type can even be `java.lang.Object` to store any instance, but that isn’t useful: wouldn’t take advantage of compiler’s type-checking

Arrays (3/4)
- Every array element is an object reference, sub-array, or base type. What real-world objects can be organized by arrays?
  - number of electoral votes by state
  - streets in Providence
  - Strings representing names or Banner IDs of people in a course
- Elements ordered sequentially by numerical index
  - in math, use subscript notation, i.e., $A_0$, $A_1$, $A_2$, ...$A_{n-1}$
  - in Java, use index inside brackets, i.e., for an array of $n$ number of students: `students[0]`, `students[1]`, ... `students[n-1]`

Arrays (4/4)
- Arrays store objects in numbered slots
  - for array of size $n$, first index is always 0, last index is always $n-1$
- Common graphical representations of arrays:

Note: 1D Arrays are called vectors, and 2D or n-D arrays are called matrices in mathematics.
### Array Examples
- **Houses on a Neighborhood Street**
  - Array size: 8
  - Array index: house number
  - Element type: house

  *Note: arrays don't need to be full (e.g., no house 0, 4, or 7)*

- **Sunlab Computers**
  - Array size: 72
  - Array index: computer number
  - Element type: computer

  *Note: could be modeled as a 2-D array (see slide 51)*

### Outline
- **Purpose**
- **Array Syntax**
- **ArrayLists**
- **Multi-Dimensional Arrays**

### Java's Syntax for Arrays (1/4)

```
<type>[] <array-name> = new <type>[<size>];
```

- Declaration
- Initialization

  *e.g., Dog[] dogArray = new Dog[101];*

- `<type>` denotes data type array holds: can be class, base type, interface, superclass, or another array (nested arrays)
  - no reserved word “array”-[] brackets suffice
- We use `new` here, because arrays are a Java **construct**
- `<size>` must be integer value greater than 0; indices range from 0 to `<size> - 1`
Java's Syntax for Arrays (2/4)

- Arrays can be local variables, so they can get declared and initialized in single statement - just like objects and base types:
  
  ```java
  Colorable[] otherColorables = new Colorable[5];
  ```

- Arrays can also be instance variables, which get declared and then initialized separately in constructor:
  
  ```java
  private Colorable[] myColorables;
  ```

  ```java
  //in constructor of class that contains the array
  this.myColorables = new Colorable[10];
  ```

Initializing Array Example

- Houses on a neighborhood street
  
  ```java
  House[] houses = new House[8];
  ```

  //next enter values in array

- Sunlab Computers
  
  ```java
  Computer[] sunlab = new Computer[72];
  ```

- Only array is initialized, not elements of array; all references are set to a default of null for Objects, 0 for ints, false for booleans, etc.
  
  ```java
  House[] houses = new House[8];
  ```

  Does not create and store 8 instances of House

Java's Syntax for Arrays (3/4)

- Accessing individual elements:
  
  ```java
  <array-name>[<index>]
  ```

  - `index` must be integer between 0 and (array size-1)
  - `result` is value stored at that index
  - `if <index> > size, or < 0,
  - `ArrayIndexOutOfBoundsException` gets thrown

  - Think of `student[i]` as the “name” of that particular student (like students) — simpler way to refer to each individual element in collection, better than having to use unique names
Accessing Array Elements Example

- Houses on a Neighborhood Street
  ```java
  House[] houses = new House[8];
  // code initializing array elements elided
  House myHouse = houses[6];
  ```

- Sunlab Computers
  ```java
  CPU[] sunlab = new CPU[72];
  // code initializing array elements elided
  CPU myCPU = sunlab[42];
  ```

Java’s Syntax for Arrays (4/4)

- An array element will work anywhere a variable would
  ```java
  // initialize first element of array of objects implementing Colorables to be a Ball
  myColorables[0] = new Ball();
  // call a method on 3rd element
  myColorables[2].setColor(Color.RED);
  // assign fourth element to a local variable
  Colorable myColorableVar = myColorables[3];
  // pass 5th as a parameter
  this.myPaintShop.paintRandomColor(myColorables[4]);
  ```

Arrays as Parameters (1/3)

- Can pass entire array as parameter by adding array brackets to type inside signature
  ```java
  public int sum(int[] numbers){ // no size declared!
      // code to compute sum of elements in the int array
  }
  ```

- Now we can do the following (somewhere else in the class that contains sum):
  ```java
  int[] myNumbers = new int[5];
  // code elided - initializes myNumbers with values, sum method
  System.out.println(this.sum(myNumbers));
  ```

Note: there is no way to tell from the use of sum that myNumbers is an array - would need to see how sum was declared to know that!
Arrays as Parameters (2/3)

- How do we determine size of array?
  - arrays have `length` as public property (not a method)
  - use special "dot" syntax to determine `length`; here we inquire it, then store it for later

```
int arrayLength = <array-name>.length;
```

Arrays as Parameters (3/3)

- How does `.length` work in actual code?

```
public int sum (int[] numbers){
    //sum all entries in array
    int total = 0;
    for (int i = 0; i < numbers.length; i++){
        total += numbers[i];
    }
    return total;
}
```

Note: for loop often used to traverse through all elements of array. Can use loop counter (i, in this case) inside the body of loop but should never reset it. Incrementing/decrementing counter is done by for loop itself!

Example: Hunger Games Tribute Selection (1/2)

- We want to store all 13 Districts from the Hunger Games one by one, using our array of Districts so we can select tributes for the Games
Example: Hunger Games Tribute Selection (2/2)

```java
// first, declare and initialize the array
District[] districts = new District[13];

// then, initialize the contents of the array
districts[0] = new District("Luxury District");
districts[1] = new District("Masonry District");
... 
districts[12] = new District("Nuclear District");

// lastly, use a loop to select the tributes
for (int i = 0; i < districts.length; i++) {
    districts[i].selectTributes();
}
```

Note: actually district 13, because arrays are indexed at 0.

ArrayIndexOutOfBoundsExceptions (1/2)

- Careful about bounds of loops that access arrays!
- Java throws `ArrayIndexOutOfBoundsException` if index is negative since sequence starts at 0
- Also throws `ArrayIndexOutOfBoundsException` if index is ≥ array size; remember that array goes from 0 to n-1

```java
District[] districts = new District[13];

districts[0] = new District("Luxury District");
districts[1] = new District("Masonry District");
...
districts[12] = new District("Nuclear District");

for (int i = 0; i < districts.length; i++) {
    districts[i].selectTributes();
}
```

ArrayIndexOutOfBoundsExceptions (2/2)

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
    Index 13 out of bounds for length 13
at (HungerGames.java:64)
```

Example of a classic "off-by-one" error!

```java
District[] districts = new District[13];
districts[0] = new District("Luxury District");
districts[1] = new District("Masonry District");
...
districts[12] = new District("Nuclear District");

for (int i = 0; i < districts.length; i++) {
    districts[i].selectTributes();
}
```
TopHat Question

Consider the sum function from slide 19:

```java
public int sum(int[] numbers)
{
    int total = 0;
    for (int i = 0; i < numbers.length; i++)
    {
        total += numbers[i];
    }
    return total;
}
```

What if the code read:

```java
i <= numbers.length?
```

A. It would wrap around and add the value at index 0 again
B. It would reach the last element of the array
C. It would raise an `ArrayIndexOutOfBoundsException`
D. None of the above

---

**for vs. for-each loop (1/4)**

- Intended to simplify most common form of iteration, when loop body gets applied to each member of collection
- How do `for-each` loop and `for` loops differ?
  - `for` loop gives access to index where item is stored
  - `for-each` loops don't have direct access to index, but can easily access item (see next example)

---

**for vs. for-each loop (2/4)**

- `for` loops were extended to `for-each` loops, which iterate over the contents of a data structure rather than indices

```
for (<type> <var>: <structure>)
{
    <loop body>
}
```

- `<type>`: class of objects stored in the `<structure>`
- `<var>`: name of current element—holds each successive element in turn; effectively local variable whose scope is loop
- `<structure>`: data structure (array or other collection) to iterate through
**for vs. for-each loop (3/4)**

- If every element needs to be iterated and loop body doesn't need element index, for-each loops suffice:

  ```java
  for (District currentDistrict : districts){
    //notice we don't need to use index to get members from Array
    currentDistrict.selectTributes();
  }
  ```

- Great advantage of for-each loops is that they don't raise `ArrayIndexOutOfBoundsExceptions`! Why?
  - Java does the indexing for you!

**for vs. for-each loop (4/4)**

- Consider this for loop:

  ```java
  for (int i = 0; i < districts.length; i++){
    if (i % 2 == 0) { //if index 'i' is even
      districts[i].selectTributes();
    }
  }
  ```

- Only want to draw tributes from districts with even index, so for-each loop wouldn't work
  - we don't execute `selectTributes()` on every element in the array; we only care about elements at specific indices

**Inserting and Deleting in Arrays (1/2)**

- Arrays are great for static data, but inserting without overwriting existing data and deleting without leaving slots empty requires explicit programmer support
- If the array contains data sorted alphanumerically, can't just insert a new item at the end (assuming there is room in the array) but must move data over to make room at the appropriate slot in the array
  - When inserting at particular index, all other elements at and after that index must get shifted right by programmer (their indices are incremented by 1) otherwise data at index of insertion would get erased and replaced
Inserting and **Deleting** in Arrays (2/2)

- When **deleting** from particular index, could leave slot empty, but more commonly, to preserve space, all other elements falling after that index get **shifted** left by programmer to fill the newly opened space (index decremented by 1). This does affect the index of all other items to the right.

Outline

- **Purpose**
- **Array Syntax**
- **ArrayLists**
- **Multi-Dimensional Arrays**

**java.util.ArrayList** (1/2)

- **java.util.ArrayList**, like arrays, hold references to **many objects of same** data type
- Another kind of **collection**, also using an index, but much easier management of making changes to array at runtime
- As name implies, has properties of both Arrays and **List** (covered later) – typically implemented "under the hood" by Java with arrays
- Differences with arrays:
  - don't need to be initialized with size – can hold an **arbitrary** and **mutable** number of references
  - are Java classes, not Java constructs, so have methods
Java.util.ArrayList (2/2)

- Why use them instead of arrays?
  - when number of elements to be held is unknown
  - storing more data in an array that's too small leads to errors
  - making array too large is inefficient, wastes memory
  - ArrayList handles update dynamics (shifting elements in memory) for you

- Why use arrays instead of array lists?
  - want something simple
  - want to use less memory (when you expect both array and array list to hold same number of elements)
  - want faster operations

Objects

- ArrayLists, like arrays, can hold any Object!
- Every class implicitly extends Object
  - every object "is an" Object
- Object is the most generic type possible
  - Object effie = new Dog();
  - Object pongBall = new CS15Ball();
  - Object cartoonPane = new Pane();

What can ArrayLists hold?

- Upside: ArrayLists store things as Object—maximum polymorphic flexibility
  - since everything is an Object, ArrayLists can hold instances of any and every class: total heterogeneity
  - easy inserting/deleting anything
- Downside: ArrayLists only store Objects:
  - only methods available are trivial ones of Object itself: equals(), toString(), and finalize()
  - typically want what an array is: homogeneous collection to store only objects of particular type (and its subtypes) AND have the compiler do type-checking for that type to enforce homogeneity
Generics! (1/2)

- Generics allow designer to write collection class A to hold instances of another class B, without regard for what class B will be. User of that class A then decides how to restrict/specialize type for that homogeneous collection.

- Constructor of the generic `ArrayList` (a collection class) provided by Java:
  ```java
  public ArrayList<ElementType>();
  ```

- Think of `ElementType` as a "type parameter" that is used as a placeholder that the user will substitute for with any non-primitive type (class, interface, array, ...)
  - primitive types: `boolean`, `int`, `double` must be special-cased – Slide 42

- Provides flexibility to have collection store any type while still having compiler help enforce homogeneity by doing type checking.

Generics! (2/2)

- With generics, `ArrayList` was implemented by the Java team to hold any `Object`, but once an instance of an `ArrayList` is created by a programmer, they must specify type. Let's create an `ArrayList` of `HungerGamesTributes` for our CS15 Hunger Games!

  ```java
  ArrayList<HungerGamesTributes> tributes = new ArrayList<>();
  ```

- We specify `HungerGamesTributes` as type that our `ArrayList`, `tributes`, can hold. Java will then replace `ElementType` with `HungerGamesTributes` in `ArrayList` method parameters and return types.

- Can think of generics as a kind of pseudo-parameter, with different syntax (the <>) since only methods have parameters, not classes. Here, `ElementType` is a pseudo-parameter and `HungerGamesTributes` is pseudo-argument.

- Generics, like classes and methods with parameters, provide generality in programming! (as does polymorphism in parameter passing)

java.util.ArrayList Methods (1/6)

```
// Note: only most important methods shown (All defined for you!)
// see Javadocs for full class
// Note: literal use of < and >, only on the constructor; most methods use the specified ElementType

public ArrayList<ElementType>();
// one of the many constructors for ArrayList class -- specialize
// by providing ElementType, just as Array has the type it // stores.

public ElementType get(int index)
// returns the object of type ElementType at that index
```
java.util.ArrayList Methods (2/6)

// two add methods with unique method signatures - method overloading
public boolean add(ElementType element)
// inserts specified element at end of ArrayList
public void add(int index, ElementType element)
    // inserts the specified element at the specified index in
    // this ArrayList, just as with arrays, causes indices of
    // elements “to the right” to be incremented - but is done automagically
public boolean remove(ElementType elem)
// remove first occurrence of specified element and returns true
// if ArrayList contains specified element
public ElementType remove(int index)
// removes the ElementType at given index and returns it

java.util.ArrayList Methods (3/6)

public int size()
// returns number of elements stored in ArrayList

public boolean isEmpty()
// returns true if ArrayList contains zero elements; false otherwise

java.util.ArrayList Methods (4/6)

- ArrayLists also have methods that access elements through search (as opposed to using an index)
  - these methods take parameter of type Object
  - but should never pass in anything besides ElementType
java.util.ArrayList Methods (5/6)

- `indexOf(ElementType elem)`
  - Finds first occurrence of specified element, returns -1 if element not in ArrayList

- `contains(ElementType elem)`
  - Returns true if ArrayList contains specified element

java.util.ArrayList Methods (6/6)

- Some other ArrayList notes...
  - Can add object in particular slot or append to end
  - Can retrieve object stored at particular index and perform operations on it
  - Can use for or for-each loop to access all objects in ArrayList
  - Shifting elements for adding/deleting from ArrayList is done automatically by Java!
    - Beware that indices past an insertion/deletion will increment/decrement respectively

ArrayList Example (1/2)

- Store an ArrayList of baking items in your pantry, using the Ingredient interface as the generic type

```java
ArrayList<Ingredient> pantry = new ArrayList<>(); // empty ArrayList
pantry.add(new Flour()); // inserts at back of list, index 0
pantry.add(new Sugar()); // inserts at back of list, index 1
pantry.add(1, new ChocolateChips()); // inserts at index 1
```

Pantry ArrayList

<table>
<thead>
<tr>
<th>Index 0</th>
<th>Index 1</th>
<th>Index 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flour</td>
<td>Sugar</td>
<td>ChocolateChips</td>
</tr>
</tbody>
</table>
**ArrayList Example (2/2)**

```java
Ingredient mySugar = pantry.get(2); // returns Sugar instance
pantry.add(new BakingPowder()); // inserts at back of list, index 3
pantry.remove(mySugar); // removes Sugar instance
pantry.remove(0); // removes Flour instance
pantry.get(2); // raises ArrayIndexOutOfBoundsException
```

**Summary of ArrayLists (1/2)**

- More flexible than arrays for insertion/deletion
  - Dynamically shifting elements and adjusting size in response to insert/delete is done automatically
- Useful methods and return types:
  - `getElement(int index)`
  - `boolean add(ElementType element)`
  - `void add(int index, ElementType element)`
  - `int indexOf(ElementType elem)` //search
  - `ElementType remove(int index)`
  - `boolean remove(ElementType elem)`
  - `int size()`
  - `boolean isEmpty()`

**Weird edge case**: To make an ArrayList of primitive types, just specify `Boolean`, `Integer`, or `Float` in the generic brackets. The `Boolean remove()` also has a weird edge case for `Integer`: you cannot use `remove(5)` to remove the first occurrence of `5`, because it will treat it as the `ElementType` for remove. This would remove whatever is at index 5. To remove an `Integer` element, use `remove(new Integer(<number>))`.

**Summary of ArrayLists (2/2)**

- Can hold heterogeneous collection of any kind of `Object`; want homogeneous collections...
- Specialize the `ArrayList` type by adding “generic” specification to a declaration or instantiation - thereby specifying two classes in one statement: the collection and the type of object it will hold and return
  ```java
  ArrayList<HungerGamesTribute> tributes = new ArrayList<>();
  ```
  - New tributes will only hold instances of type `HungerGamesTribute`
- Remember to use literal `<>` for specialized type!
**TopHat Question**

Which of the following uses an `ArrayList` correctly?

A. `ArrayList<HungerGamesTribute> tributes = new ArrayList<>();
   HungerGamesTribute heroicTribute = new HungerGamesTribute();
   tributes.add(heroicTribute);

B. `ArrayList<ElementType> tributes = new ArrayList();
   HungerGamesTribute wimpyTribute = tributes[0];`

C. `ArrayList<ElementType> tributes = new ArrayList<>();
   HungerGamesTribute backstabbingTribute = tributes.first();`

D. `ArrayList<String> tributes = new ArrayList<>();
   HungerGamesTribute crazyTribute = new HungerGamesTribute();
   tributes.add(crazyTribute);`

---

**ConcurrentModificationExceptions**

```java
public static void main(String[] args)
    Arraylist<hungergamestribute> tributes = new Arraylist<>();
    tributes.add(new HungerGamesTribute("Cannon");
    tributes.add(new HungerGamesTribute("Lexi");
    tributes.add(new HungerGamesTribute("Anastasio");
    tributes.add(new HungerGamesTribute("Allie");
    tributes.add(new HungerGamesTribute("Sarah");

    for (hungergamestribute t : tributes)
    if (!t.getName().equals("Sarah");
       tributes.remove(t);
    }
```  

When trying to modify an `ArrayList` while iterating through it with a `for-each` loop, you will get a `ConcurrentModificationException`.

- Adding and removing cannot be done within a `for-each` loop because of the shifting of the elements in the list that Java does in response to an add or remove.
- Note: this is important for DoodleJump! We’ll go over this issue in detail during the project help slides and in section

---

**Outline**

- Purpose
- Array Syntax
- ArrayLists
- Multi-Dimensional Arrays
Multi-Dimensional Arrays

- Modeling chessboard:
  - not linear group of squares
  - more like grid of squares
- Multi-dimensional arrays are arrays of arrays of...
- Can declare array to be 2 (or more) dimensions, by adding more brackets
  - one pair per dimension
  - 2-D: `int [][] grid = new int [a][b];`
  - 3-D: `int [[[] cube = new int [[x][y][z]];`
  // a, b, x, y, z are ints whose values are set elsewhere

2-Dimensional Array Examples (1/2)

- Pixel Array
  - 2-D Array size: width by height
  - array indices: x, y
  - element type: RGB color
  - `Pixel[][] MSFTLogo = new Pixel[x][y];`
- Connect Four
  - 2-D Array size: 7 by 6
  - array indices: row, column
  - element type: checker
  - `Checker[][] connect4 = new Checker[6][7];`

2-Dimensional Array Examples (2/2)

- The Sunlab
  - 2-D Array size: 8 by 10 (approx.)
  - array indices: row, column
  - element type: computer
  - `Computer[][] sunlab = new Computer[10][8];`
Let's say we want to represent this grid of numbers:

How do we want to represent this grid? There are two equally valid options:

Multi-dimensional arrays in Java do not make a distinction between rows or columns:
- think about 1D arrays – it doesn’t really matter if we call it a “row” or a “column”
- can think of arrays as ordered sequences of data stored in contiguous positions in memory - no intrinsic geometry/layout implied
Ways to Think About Array Storage (2/3)

- Two visualizations of two-dimensional array (called ballArray) are equally valid. You can choose either for the organization of your array.

- Make sure there's consistency in the way you index into your 2D array throughout your program!
  - since the elements are not stored in a specific order, the way that we insert elements and initialize and index into our array determines the order

- Column of Rows:
- Row of Columns:

Ways to Think About Array Storage (3/3)

- The choice between row-major and column-major organization can sometimes be arbitrary
  - Connect 4, a large carton of eggs, etc.
- However, sometimes one will make more sense or simplify your program based on what you are trying to achieve
- Can Storage example
  - goal: use array to keep track of the number of each type of can
    - makes most sense to use column-major organization
      - each column would be a sub-array of cans of the same type
      - slots within each column are either null (empty) or hold a can
      - can count number of each type by checking to see how many entries are full (or not null) in each sub-array (column)
  - For a table of entries (e.g. student rows, course grades cols) use row major order, while for GetPixel (x, y) use column major order

TopHat Question

Here’s a grid of colored golf balls in column major order. What index is the light blue golfball in?

A. ballArray[2][3]
B. ballArray[2][1]
C. ballArray[3][2]
D. ballArray[1][2]
Common Array Errors - Watch Out! (1/2)

- Cannot assign a scalar to an array

```
int[] myArray = 5;  // X
```

  - 5 is not an array
  - To initialize array elements, must loop over array and assign values at each index. Here we assign 5 to each element:

```
int[] myArray = new int[20];  // Initializes array, not elements
for (int i=0; i < myArray.length; i++){
    myArray[i] = 5;
}
```

Common Array Errors - Watch Out! (2/2)

- Cannot assign arrays of different dimensions to each other

```
int[] myIntArray = new int[23];
int[][] my2DIntArray = new int[2][34];
myIntArray = my2DIntArray;  // X
```

- Doing so will result in this error:
  
  "Incompatible types: Can't convert int[] to int[][]"

- Similar message for assigning arrays of mismatched type

- Take note that Java will automatically resize an array when assigning a smaller array to a larger one

SciLi Tetris: Loops and Arrays Writ Large

- In 2000, Tech House constructed then the largest Tetris game on the Scili – the Woz flew out to play it!

- 5 months of work: 11 custom-built circuit boards, a 12-story data network, a Linux PC, a radio-frequency video game controller, and over 10,000 Christmas lights – see http://bastilleweb.techhouse.org/

- Video: https://www.youtube.com/watch?v=6kIRWoz9z1b&lc=21s

- Article: http://news.bbc.co.uk/2/hi/science/nature/718009.stm
Announcements

• Cartoon deadlines
  - early handin: tonight, 10/19
  - on-time handin: Saturday, 10/21
  - late handin: Monday, 10/23
  - remember to tackle Minimum Functionality before trying any Bells & Whistles!

• DoodleJump partner form due Saturday night
  - if you don’t fill it out, you’ll be assigned a random partner on no basis
  - if choosing your own partner, you must both fill it out with the correct logins

Privacy and Surveillance II: Cases and Protective Laws

Case Study: Reproductive Health Data Tracking
FTC Finalizes Order with Flo Health, a Fertility-Tracking App that Shared Sensitive Health Data with Facebook, Google, and Others

Post-Roe, prosecutors can seek unprotected reproductive health data

Citizens, Not the State, Will Enforce New Abortion Law in Texas
Case Study: Private Cameras and Policing

Ring, Google and the Police: What to Know About Emergency Requests for Video Footage

Amazon provided police with Ring video footage without user consent or a warrant under “emergency requests.”

Google does the same with Nest footage.

Breaking News: Transport Security Administration Tracking

The mobile driver's license (mDL) is comprised of the same data elements that are used to produce a physical drivers license, however, the data is transmitted electronically - American Association of Motor Vehicle Administration

Aadhaar Data Breach — How Sensitive Data Of 1.3 Billion Indians Was Compromised

Protective Laws
Data breaches must be reported to governments within 72 hrs. Limits on how data can be collected and what is collected. “Strengthen the ‘right to be forgotten’” — process to remove your data from services completely.

May 2018 - General Data Protection Regulation (GDPR)
Set of privacy regulations in the EU meant to harmonize laws between member countries
• Limits on how data can be collected and what is collected
• “Strengthen the ‘right to be forgotten’” — process to remove your data from services completely

January 2020 - California Consumer Privacy Act
• “The Golden State officially has the strongest consumer data protections in the US” [WIRED, 2020]. Applies to businesses established in California:

Sept. 15, 2022 - California Age Appropriate Design Code
• Requires online platforms to proactively consider how their product design impacts the privacy and safety of children and teens in California. Companies must:

Photo Credit: The Conversation
Image source: termly.io
Image source: Human Tech
How is Tech Policy Shaped?

- In 118th (current) Congress of 535 members, 4 scientists, 9 engineers, 4 software company executives
- The Internet Association: industry players that make tech policy suggestions
- Lobbying + vacuum of knowledge around issues — often is just what is best for industry!
- Consider working in tech policy!

In 2021, Brown CS Professor Suresh Venkatasubramanian was appointed to the White House Office of Science and Technology Policy, advising on matters relating to fairness and bias in tech systems.

Proposal 2022 - American Data Protection and Privacy Act

- "This bill establishes requirements for how companies, including nonprofits and common carriers, handle personal data, which includes information that identifies or is reasonably linkable to an individual." (U.S. Library of Congress, 2022)
- Limit the collection and transfer of personal data
- Additional protections for minors
- Means to opt out of targeted advertising
- More robust company cybersecurity
- Right to more control over personal data
This Week’s SRC Discussion!
More on GDPR and ADPPA

Appendix: 2D Array Example

Example: Size of 2-D Arrays
2D Arrays Example (1/2)

- Let's build a checkerboard with alternating black and white squares, using JavaFX
- Each square has a row and column index
- Let's use row-major order
  - access any square with `checkerboard[rowIndex][colIndex]`
- JavaFX `Rectangle`'s location can be set using row and column indices, multiplied by square width factor
  - row indicates Y values, column indicates X value

```
// instantiate a Pane and initialize the checkboard 2D array
Pane myPane = new Pane();
Rectangle[][] checkerboard = new Rectangle[Constants.NUM_ROWS][Constants.NUM_COLS];
// loop through row and column indices (for each col in each row…)
for (int row = 0; row < checkerboard.length; row++) {
  for (int col = 0; col < checkerboard[row].length; col++) {
    // instantiate rectangle, setting Y/X loc using row/col indices
    Rectangle rect = new Rectangle(col * Constants.SQ_WIDTH, row * Constants.SQ_WIDTH, Constants.SQ_WIDTH, Constants.SQ_WIDTH);
    // alternate black and white colors
    if ((row + col) % 2 == 0) {
      rect.setFill(Color.BLACK);
    } else {
      rect.setFill(Color.WHITE);
    }
    myPane.getChildren().add(rect); // graphically add the rectangle
    checkerboard[row][col] = rect; // logically add the rectangle
  }
}
```