Lecture 13

Arrays and ArrayLists

C	S	C	I	0	1	5	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Outline

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

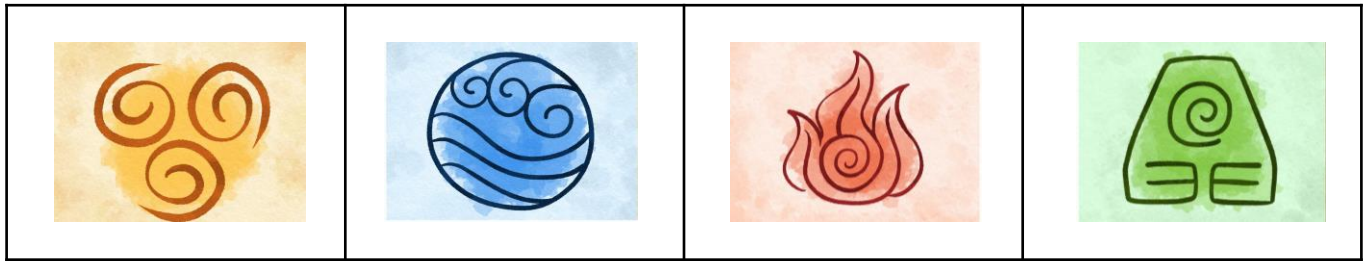
Why Use Arrays? (1/2)

- 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
  - 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**
  - allows for simple communication with multiple elements
  - 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 4 instances of `AvatarElement`?
  - store all `AvatarElement`s in array for easy access (to make sure we have the right number of elemental arts mastered by The Avatar!)



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

Your lovely TAs

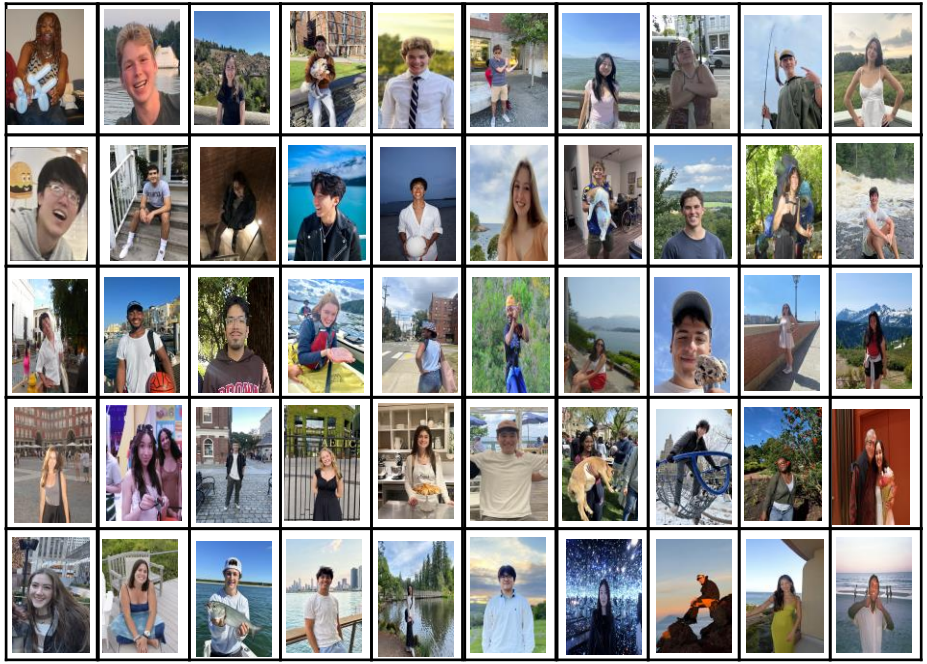
- We want access to all 45 UTAs (and 5 HTAs)!
  - Aisosa, A.J., …, Yabeke
- Could use instance variables:

```
public class CS15TAs {

    private TA aisosa;
    private TA aj;
    //other TAs elided
    private TA yabeke;

}
```

- Can't access 50 instance variables very easily
  - what if we wanted to access CS15 TAs from fall 2023, 2022, 2021, …



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:

```
taArray[index]
```

    - the `index` of array is always of type `int`
  - neither base type nor class, but Java **construct**
    - use `new` to initialize an array (even though it's not a class!)
    - does not invoke constructor like a class
    - designed for storage and runtime efficiency for large datasets

Arrays (2/4)

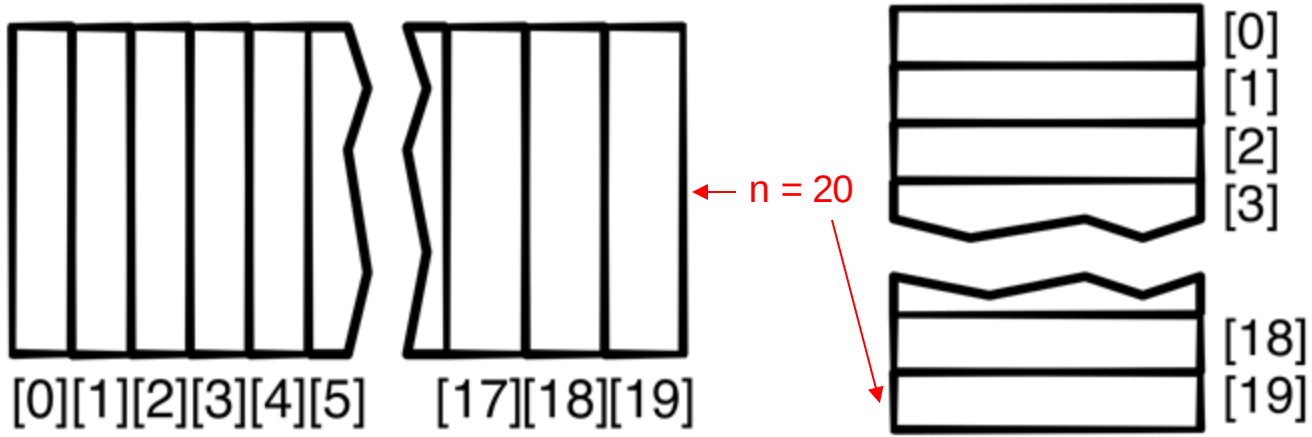
- Arrays only hold elements of specified type
  - when declaring arrays, state **type** of object it stores:
    - base type
    - class
    - sub-arrays (for multi-dimensional arrays – soon)
    - for max polymorphic flexibility, interface or superclass
  - type can even be `java.lang.Object` to store 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
  - `String`s 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:
    `taArray[0]`, `taArray[1]`,... `taArray[n-1]`

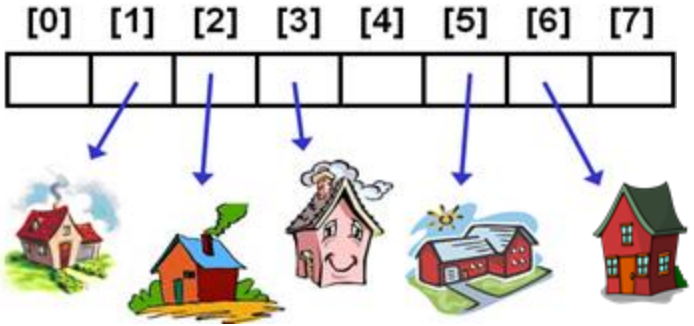
Arrays (4/4)

- 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: 1-D Arrays are called vectors, and 2-D 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 declaration "`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:

```
Colorable[] otherColorables = new Colorable[5];
```

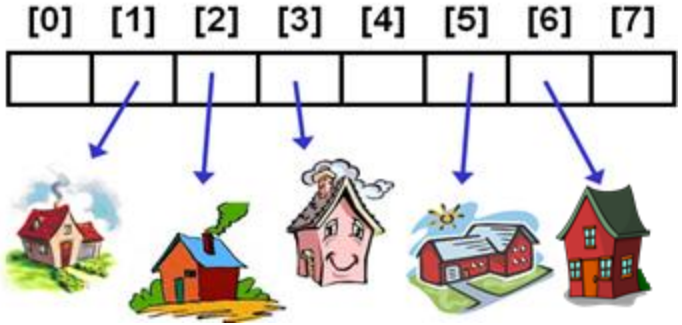
- Arrays can also be instance variables, which get declared and then initialized separately in constructor:

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

Initializing Array Example

- Houses on a neighborhood street

```
House[] houses = new House[8];
//next, enter values into array
```



- Sunlab Computers

```
Computer[] sunlab = new Computer[72];
```

- Only *array* is initialized, not *elements* of array; all references are set to a default of `null` for `Object`s, `0` for `int`s, `false` for `boolean`s, etc.

```
House[] houses = new House[8];
```

Does not create and store 8 instances of House

Java's Syntax for Arrays (3/4)

Note: some other languages allow an arbitrary value for the lower bound, but not Java!

- Accessing individual elements:

```
<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

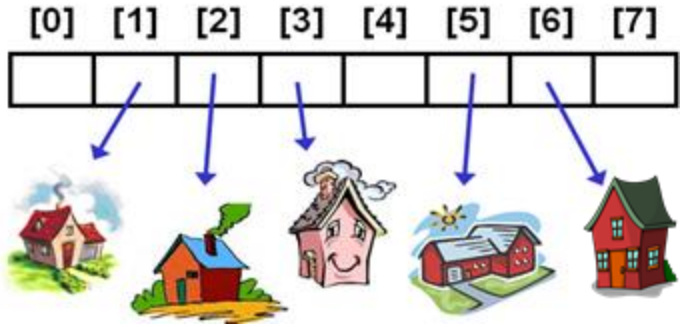
- Think of `student[i]` as the "name" of that particular student (like $student_i$) – simpler way to refer to each individual element in collection, better than having to use unique names

Accessing Array Elements Example

- Houses on a Neighborhood Street

```
House[] houses = new House[8];
//code initializing array elements elided
House myHouse = houses[6];
```

<array-name>[<index>]



- Sunlab Computers

```
Desktop[] sunlab = new CPU[72];
//code initializing array elements elided
Desktop myDesktop = sunlab[42];
```

<array-name>[<index>]

Java's Syntax for Arrays (4/4)

- An array element will work anywhere variable would

```
// initialize first element of array of objects implementing Colorables to be a Ball
myColorables[0] = new Ball();
// call a method on third element
myColorables[2].setColor(Color.RED);
// assign fourth element to local variable
Colorable myColorableVar = myColorables[3];
// pass fifth as 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

```
public int sum(int[] numbers){ //no size declared-late binding
    //code to compute sum of elements in the int array
}
```

- Now we can do the following (somewhere else in the class that contains `sum`):

```
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 this use of `sum` that `myNumbers` is an array - would need to see how `sum` and `myNumbers` were 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: Element Bender Selection (1/2)

- We want to store the four Elements using our array of type `AvatarElement`s so we can select Element benders one-by-one for The Avatar's mission

Example: Element Bender Selection (2/2)

```
// first, declare and initialize the array
AvatarElement[] avatarElems = new AvatarElement[4];

// then, initialize the contents of the array
avatarElems[0] = new AvatarElement(“Water”);
avatarElems[1] = new AvatarElement(“Earth”);
...
avatarElems[3] = new AvatarElement(“Air”);

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

Note: actually element 4, 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***

Code from previous slide

```
// first, declare and initialize the array
AvatarElement[] avatarElems = new
AvatarElement[4];

// then, initialize the contents of the array
avatarElems[0] = new AvatarElement(“Water”);

avatarElems[1] = new AvatarElement(“Earth”);
...
avatarElems[3] = new AvatarElement(“Air”);

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

ArrayIndexOutOfBoundsExceptions (2/2)

Example of a classic “off-by-one” error!

In Terminal:

```
Exception in thread “main”
java.lang.ArrayIndexOutOfBoundsException:
Index 4 out of bounds for length 4
    at (Avatarverse.java:64)
```

Note: The error tells you which index is throwing the error. Here, it is attempting to access the element at index=4, but our largest index of an array of size 4 is n-1 or, in this case, 3.

```
// first, declare and initialize the array
AvatarElement[] avatarElems = new
AvatarElement[4];

// then, initialize the contents of the array
avatarElems[0] = new AvatarElement(“Water”);

avatarElems[1] = new AvatarElement(“Earth”);
...
avatarElems[3] = new AvatarElement(“Air”);

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

TopHat Question

Join Code: 316062

Consider the sum function from slide 19, slightly modified:

What will happen?

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

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>
}
```

Can make up any arbitrary name for `<var>` *field, just like when we declare a variable and choose its name*

`<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:

```
for (AvatarElement currentAvatarElem: avatarElems){
    //notice we don't need to use index to get members from Array
    currentAvatarElem.selectElementBenders();
}
```

- 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)

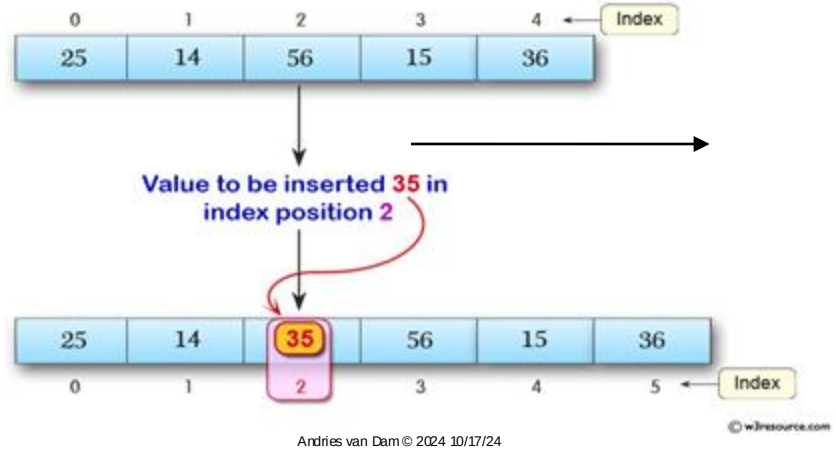
- Consider this `for` loop:

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

- Only want to draw Element benders from array elements with even index, so `for-each` loop wouldn't work
  - we don't execute `selectElementBenders()` on every element in the array; we only care about elements at specific indices
  - more generally, if you want to use the *index* inside of the loop, can't use for-each

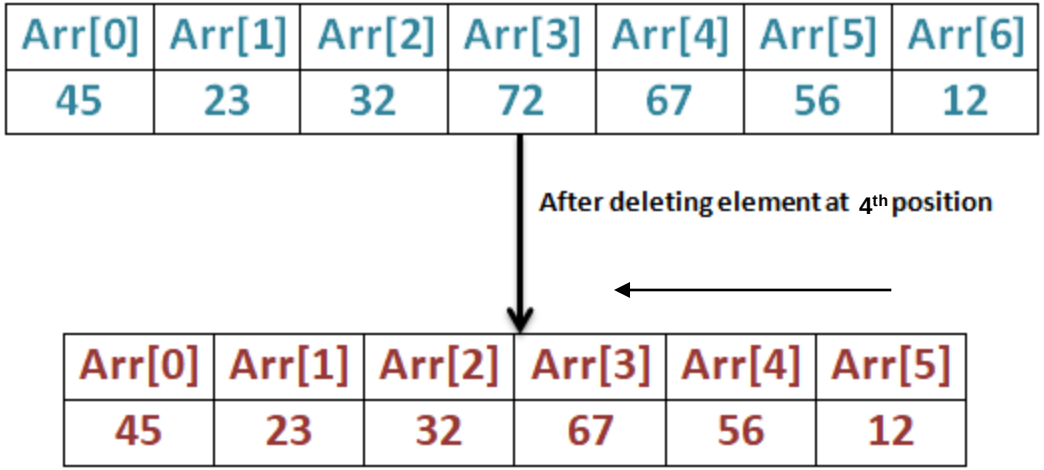
Inserting and Deleting in Arrays (1/2)

- 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 be 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). As with insertion, 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`s, 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 `Array`s and `List`s (covered later) – typically implemented ”under the hood” by Java with `Array`s
- 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
  - Trying to store 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 `ArrayList`s?
  - want something simple
  - want to use less memory (when you expect both array and `ArrayList`s to hold same number of elements)
  - want faster operations

Objects

- `ArrayList`s, 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 `ArrayList`s hold?

- Upside: `ArrayList`s store things as `Object`— maximum polymorphic flexibility
  - since **everything** is an `Object`, `ArrayList`s can hold instances of any and every class: total heterogeneity
  - easy inserting/deleting **anything**
- Downside: `ArrayList`s only store `Object`s:
  - 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:

```
public ArrayList<Type>();
```

- Think of `Type` 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 46
- 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` is implemented 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 `ElementBenders` for our CS15 Avatar Universe!

```
ArrayList<ElementBender> benders = new ArrayList<>();
```

- We specify `ElementBender` as type that our `ArrayList`, `benders`, can hold. Java will then replace `Type` with `ElementBender` 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, `Type` is a pseudo-parameter and `ElementBender` 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 Type */
public ArrayList<Type>()
/* one of the many constructors for ArrayList class – specialize
 * by providing Type, just as Array has the type it
 * stores. */
ArrayList<Type> name = new ArrayList<>();
// sample of how to declare an ArrayList


public Type get(int index)
// returns the object of type Type at that index
```

Note: think of < and > as meaning "of type"

java.util.ArrayList Methods (2/6)

```
//two add methods with unique method signatures - method overloading

public boolean add(Type element)

//inserts specified element at end of ArrayList

public void add(int index, Type 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(Type element)
/* remove first occurrence of specified element and returns true
 * if ArrayList contains specified element */

public Type remove(int index)

//removes the Type at given index and returns it
```

Note: indexing is not that useful because the indices change with editing

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)

- `ArrayList`s 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 `Type`

java.util.ArrayList Methods (5/6)

```
public int indexOf(Type elem)
/* finds first occurrence of specified element, returns -1 if
 * element not in ArrayList */

public boolean contains(Type elem)
//return 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 automagically 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 argument

```
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

size = 3

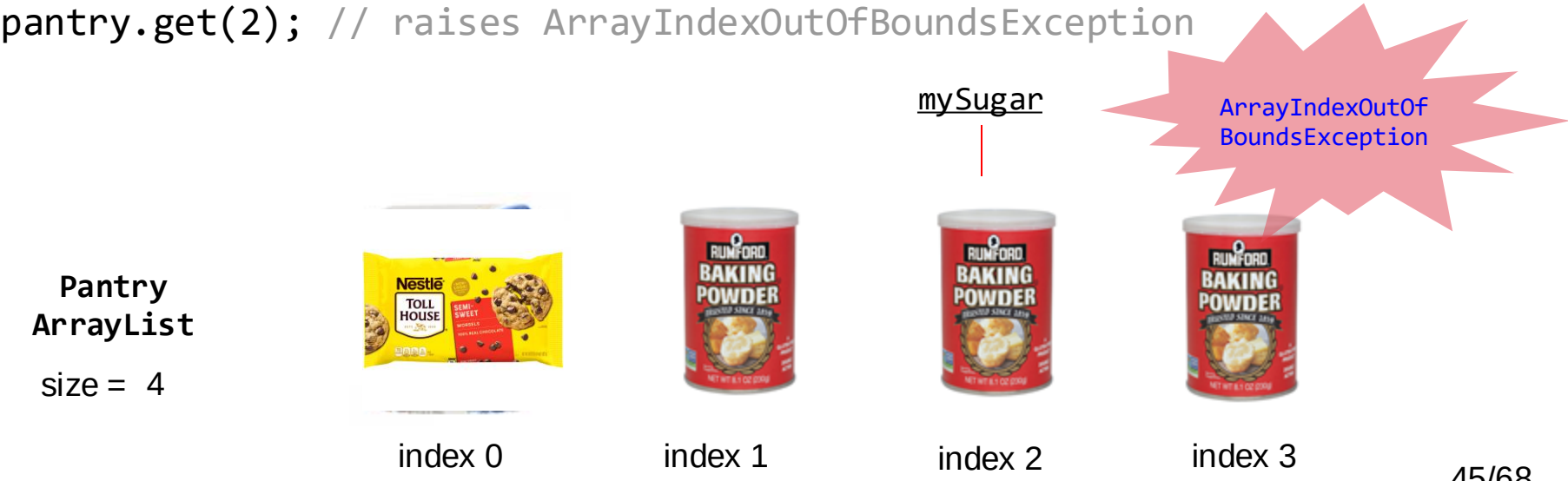
index 0

index 1

index 2

`ArrayList` Example (2/2)

```
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 all done automagically
- Useful methods and return types:
  - `Type get(int index)`
  - `boolean add(Type elem)`
  - `void add(int index, Type elem)`
  - `int indexOf(Type elem) //search`
  - `Type remove (int index)`
  - `boolean remove (Type 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 Integers: you cannot use `remove(5)` to remove the first occurrence of `5`, because it will treat it as the `Type` 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

```
ArrayList<ElementBender> benders = new ArrayList<>();
```

Now `benders` will only hold instances of type `ElementBender`

- Remember to use literal `<>` for specialized type!

TopHat Question

Join Code: 316062

Which of the following uses an `ArrayList` correctly?

A.
```
ArrayList<AirBender> airBenders = new ArrayList<>();
AirBender heroicAirBender = new AirBender();
airBenders.add(heroicAirBender);
```

B.
```
ArrayList<Type> airBenders = new ArrayList;
AirBender youngAirBender = airBenders[0];
```

C.
```
ArrayList<Type> airBenders = new ArrayList<>();
AirBender sillyAirBender = airBenders.first();
```

D.
```
ArrayList<String> airBenders = new ArrayList<>;
Airbender expertAirBender = new AirBender();
airBenders.add(expertAirBender);
```

ConcurrentModificationExceptions

```
public static void main(String[] args) {
    ArrayList<AirBender> airBenders = new ArrayList<>();
    airBenders.add(new AirBender("Ilan"));
    airBenders.add(new AirBender("Grace"));
    airBenders.add(new AirBender("Chloe"));
    airBenders.add(new AirBender("Karim"));
    airBenders.add(new AirBender("Sarah"));

    for (AirBender a: airBenders){
        if(!a.getName().equals("Chloe")){
            airBenders.remove(a);
        }
    }
}
```

In Terminal:

```
Exception in thread "main":
java.util.ConcurrentModificationExc
eption
at (App.java:13)
```

- 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

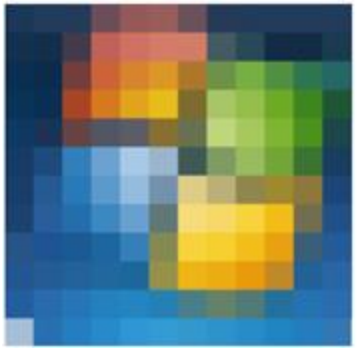
Multi-Dimensional Arrays

- Modeling chess board:
  - 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];`

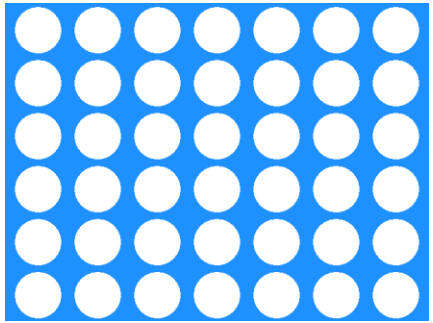
```
// a, b, x, y, z are ints whose values are set elsewhere
```

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];`

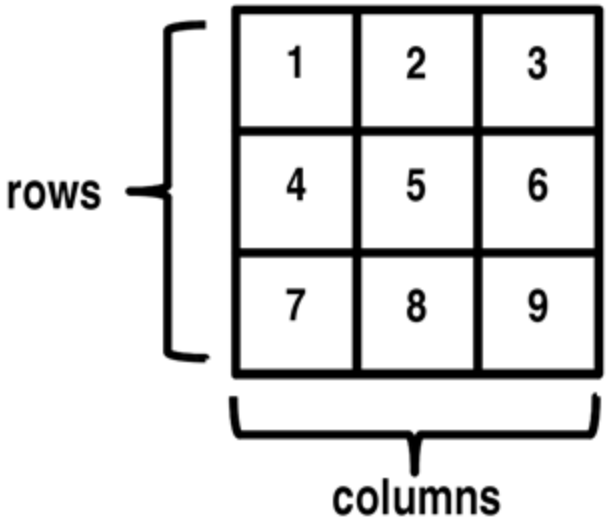


2-Dimensional Array Examples (2/2)

- The Sunlab
  - 2-D Array size: 8 by 10 (approx.)
  - array indices: row, column
  - element type: computer
  - `Computer[][] sunlab = new Computer[10][8];`

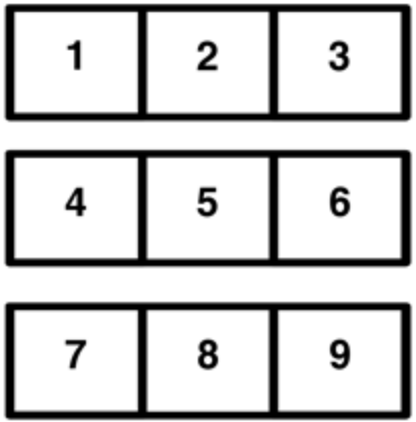
Representing Multi-Dimensional arrays (1/2)

- Let’s say we want to represent this grid of numbers:

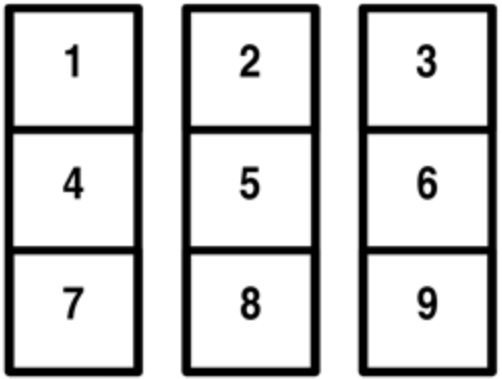


Representing Multi-Dimensional arrays (2/2)

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



Array of rows



Array of columns

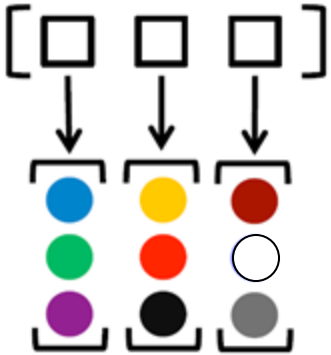
Ways to Think About Array Storage (1/3)

- Multi-dimensional arrays in Java do **not** make a distinction between rows or columns
  - think about 1D array – 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)

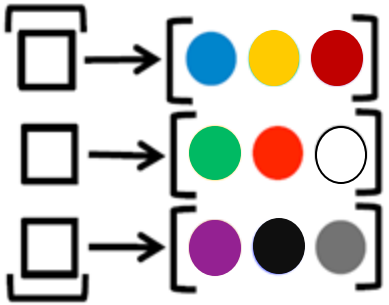
- Two visualizations of two-dimensional array (called ballArray) are equally valid. You can choose either for the organization of your array.

Row of Columns:



column-major order, i.e., first index is column index (e.g., purple ball is at array[0][2] – column 0, row 3)

Column of Rows:

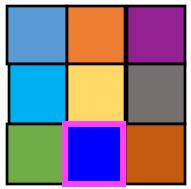


row-major order, i.e., first index is row index (e.g., purple ball is at array[2][0] – row 3, column 0)

- Make sure there's consistency in the way you index into your 2-D 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

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, here)
- 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

Join Code: 316062

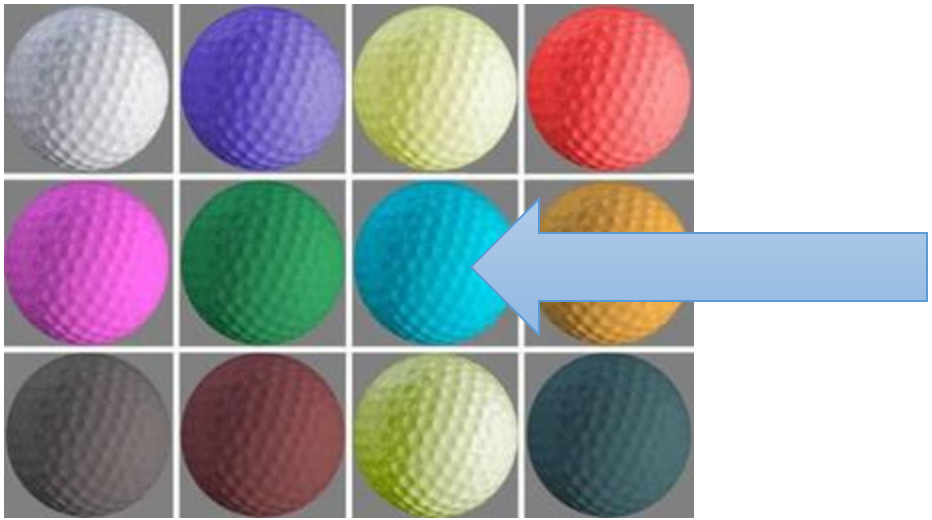
Here's a grid of colored golf balls in **column major** order. What index is the light blue golf ball in?

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;
```

  - 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;
```

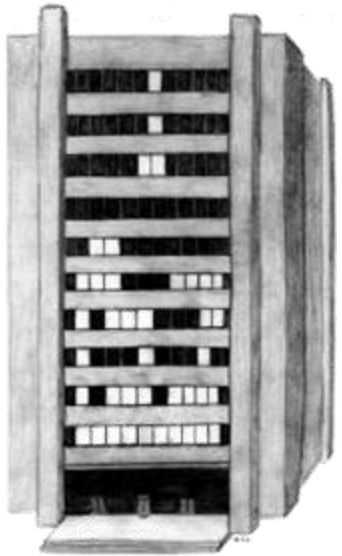
- Doing so will result in this error:

```
“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=tkIRWoo9qrU&t=21s
- Article: http://news.bbc.co.uk/2/hi/science/nature/718009.stm
- Also done in May 2024! More info here: https://scilitetris.kazar4.com/index.html



Example: Size of 2-D Arrays

```
public static final int NUM_ROWS = 10; // defined in Constants
public static final int NUM_COLS = 6; // defined in Constants

public void practice2DArrays() {
    /* deciding which is row and which is column index is
     * arbitrary but must be consistent!!! */
    String[][] myStringArray = new String[NUM_ROWS][NUM_COLS];
    int numRows = myStringArray.length;
    int numCols = myStringArray[0].length;
    System.out.println("My array has " + numRows * numCols + " slots in total!");
}
```

`array.length` gives size of first dimension (you decide whether you want row or column), and `array[0].length` gives size of second dimension

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

2D Arrays Example (2/2)

```
// 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[0].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
    }
}
```

Announcements (1/2)

- Cartoon deadlines
  - **early handin:** tonight, 10/17
  - **on-time handin:** Saturday, 10/19
  - **late handin**: Monday, 10/21
  - remember to tackle Minimum Functionality before trying any Bells & Whistles!
- DoodleJump partner form due Saturday night
  - **solo form** due tonight
  - 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

Announcements (2/2)

- Class diagram change!
  - check out Ed for more information
- Code debriefs are out – check e-mail
  - selection is random, so don’t panic if you didn’t receive one
- Cartoon Mini-Assignment checkoffs 8-9pm in CIT 210
  - Last chance to receive credit

SRC Extra Credit Discussion I

- Monday 10/21 @ 6:00-8:00pm
  - Smitty-B 106
- We will be screening and discussing *The Social Dilemma*
- Participation will give you an extra 1 point on your final grade
- Participation includes:
  - Engaging with the discussion
  - Filling out a brief post-discussion Google Form
- Please sign up via Ed **by 10/20!**

This week…

Hungry for Energy, Amazon, Google and Microsoft Turn to Nuclear Power

Large technology companies are investing billions of dollars in nuclear energy as an emissions-free source of electricity for artificial intelligence and other businesses.

The Vogtle nuclear power plant in Waynesboro, Ga. The last two nuclear units at Vogtle ran tens of billions of dollars over budget and were years late in completion.
Kendrick Brinson for The New York Times

[Big Tech & Nuclear Power - NYT]

Amazon signs agreements for innovative nuclear energy projects to address growing energy demands

3 min

October 16, 2024

Written by Amazon Staff

[Amazon nuclear energy projects]

Google

Google to buy nuclear power for AI datacentres in 'world first' deal

Tech company orders six or seven small nuclear reactors from California's Kairos Power

[Google nuclear power | The Guardian]

Privacy I: A Brief History

Topics in Socially Responsible Computing

Early American Surveillance

1908

Federal Bureau of Investigations (FBI)

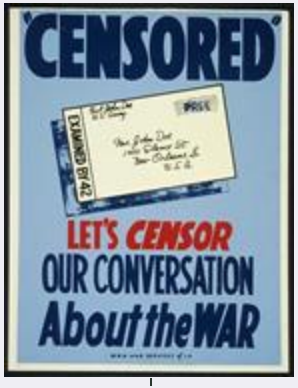
Central Intelligence Agency (CIA)

1947

1952

National Security Agency (NSA)

Import Legislation



1917 | 1918

Sedition Act &
Espionage Act

Foreign Intelligence
Surveillance Act

1978

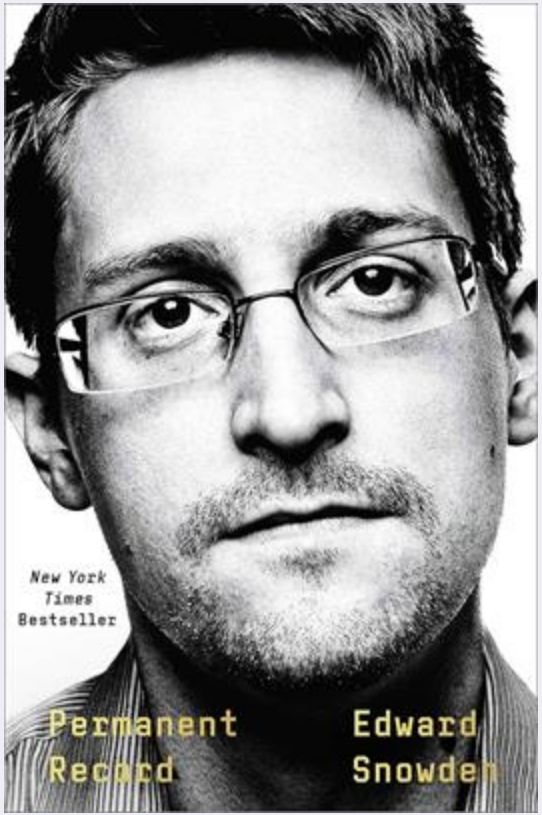
2001

Patriot Act

Protect America Act &
FISA Amendments

2007 | 2008

Whistleblowers: Edward Snowden

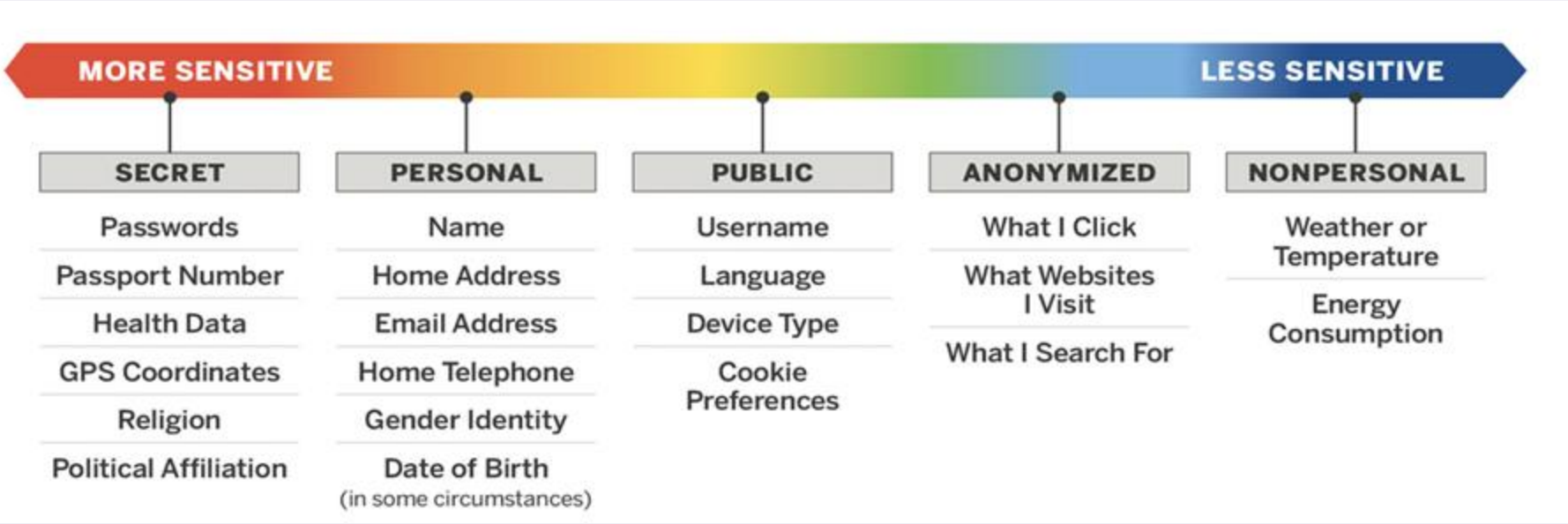


- former NSA contractor
- In 2013, leaked classified documents revealing extensive global surveillance programs
- Violated ‘Espionage Act of 1917’
- **Critique:** did ‘grave damage’ to American intelligence capacities
- **Impact:** revelations led to reforms and raised public awareness about privacy rights in the digital age

Erosion of Privacy in the Digital Age

- Privacy by default → constant data sharing
- Mass government surveillance programs
- Loss of user control & informed consent
- Surveillance capitalism & behavioral profiling
- Frequent data breaches & insecure systems

Privacy as a Spectrum



Source: Center for International Media Assistance