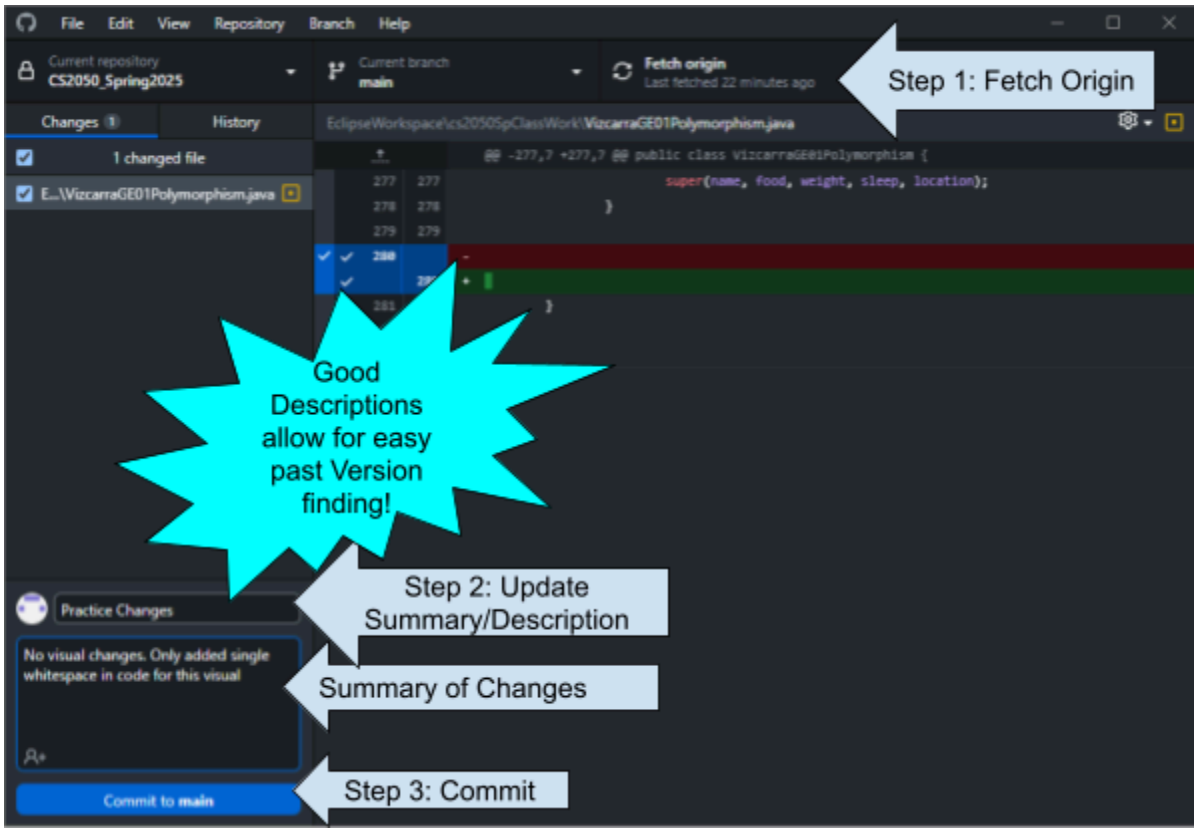
CS2050 Technical Document

Environment Setup Eclipse, Git and Github

- Tutorial to set up Eclipse and Github Desktop

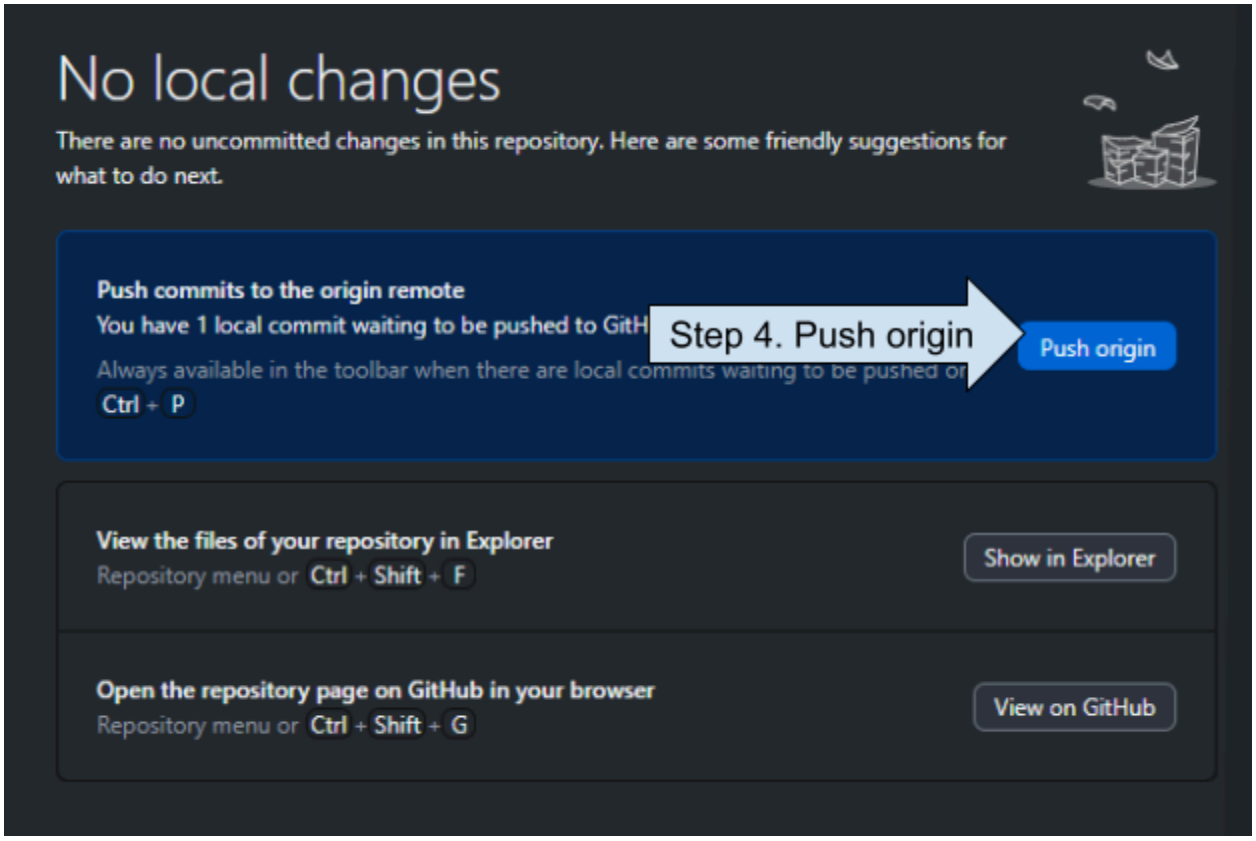
How to version code using Github Desktop

1. Open github Desktop. Press “Fetch origin.”



2. For each change, write a summary and description

3. Press “Commit to main”
    a. This saves your changes locally as a new version of the file
4. Lastly, click “Push origin”
    a. This saves the new version to GitHub

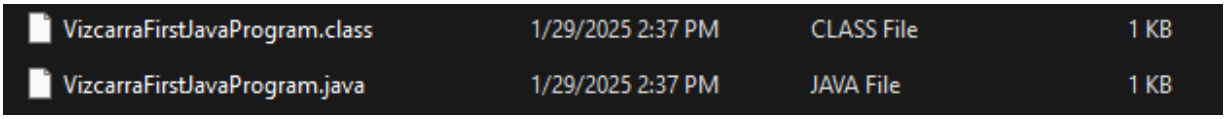


Creating in Eclipse

- Tutorial to create a new Java Project
- Tutorial to create a new Java Class File

Parts of a Java Program

Source Code: The code we write in our IDE,
Byte Code: Translation of the source code that can be read and executed by the JVM
Editing: Within our IDE, we can make changes to our code before compiling it
Compiling: When we compile a java file, it generates a class file. This class file contains the bytecode
Running: When we run our class file, the bytecode is being read and executed by the JVM



Modular Programming terms

Test Driven Development (TDD)

Test driven development is an Agile methodology that describes a programming approach where you make unit tests, and design and create code based on those tests. Reduces errors and improves maintainability

1. Break the problem into smaller parts.
2. make unit tests for each problem
3. design and write code for first test case.
4. After writing code to pass the test case, repeat with the next test case using the code you just made. make changes to code as needed to pass new test

Test case examples

Case	Precondition	Postcondition
Regular	{3,5,4,2,1}	{1,2,3,4,5}
Empty	{}	{}
Single Element	{1}	{1}
Already Sorted	{1,2,3,4,5}	{1,2,3,4,5}
Reverse Sorted	{5,4,3,2,1}	{1,2,3,4,5}
Duplicates	{1,3,2,4,1}	{1,1,2,3,4}

Classes vs Objects

- More information on classes and objects

Class: Blueprint for making an object

- Defines an object's state and behavior

```
class SimpleRectangle {
    private double length;
    private double width;
```

```
class SimpleRectangle {
    private double length;
    private double width;

    public SimpleRectangle() {
        this.length = 1;
        this.width = 1;
    }

    public SimpleRectangle(double newLength, double newWidth) {
        setLength(newLength);
        setWidth(newWidth);
    }

    public double getArea() {
        return length*width;
    }
```

Class name

State:
Data members

Constructors

Overloaded!

Behavior:
Methods

State

- represented through the class's data members
- It is things an object *knows*

Behavior

- is represented through a class's methods
- It is things is object *does*

```java
public double getArea() {
    return length*width;
}

public double getPerimeter() {
    return 2 * (length + width);
}
```

Object

Object: Instantiation of a class

- Created using new operator
- The arguments passed will determine with constructor is used

```java
SimpleRectangle rectangle1 = new SimpleRectangle();
SimpleRectangle rectangle2 = new SimpleRectangle(-3,-2);
```

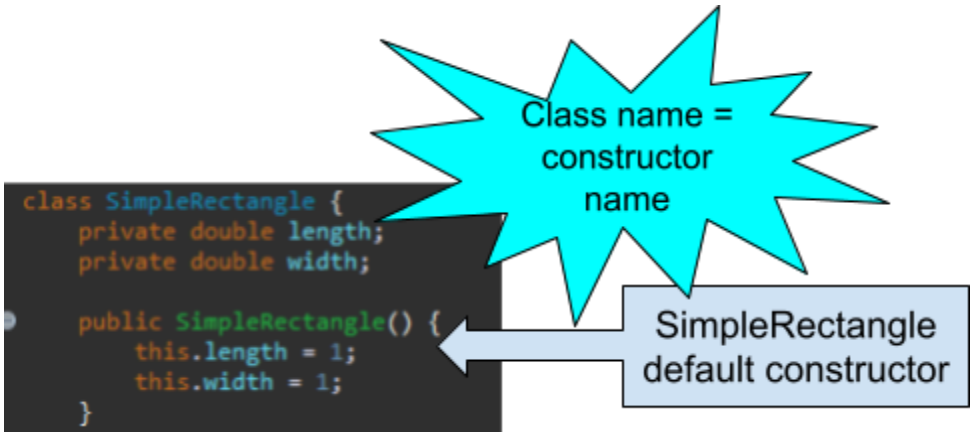
Uses default constructor

Uses SimpleRectangle(int firstInt, int secInt) constructor

Constructors

Constructor: special method used to create objects

- Has no return method, not even void
- Has same exact name as class name
- Invoked differently than regular method



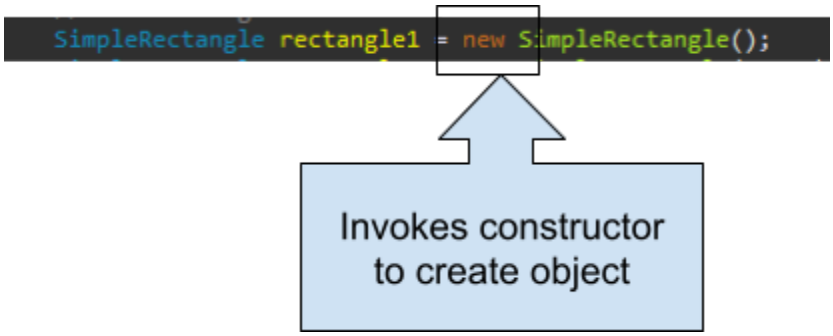
- Constructors have the exact same name (same spelling and capitalization) as their class name

Constructor vs regular method

- More information on constructors vs regular methods

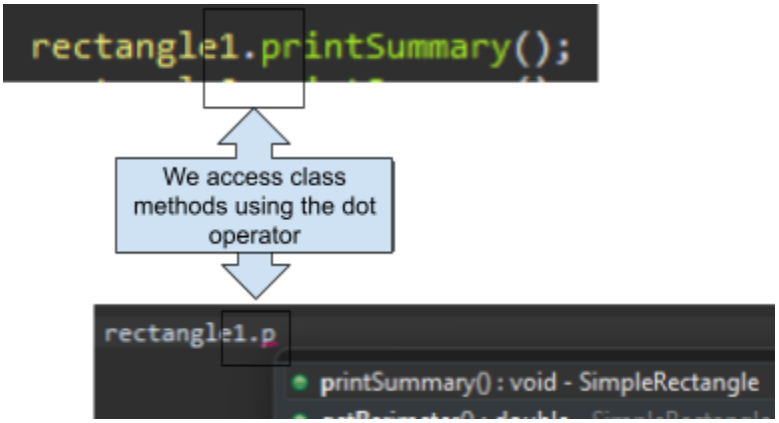
Invoking constructors

- Invoked using new operator



Invoking regular methods

- We call instance methods by using dot operator

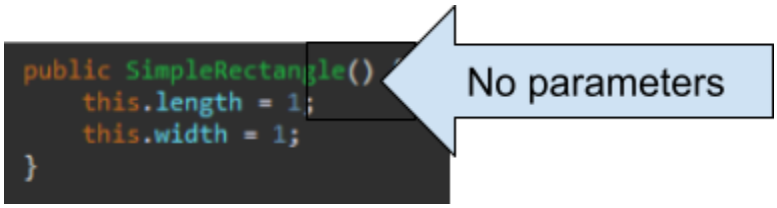


-

Default constructor vs overloaded constructor:

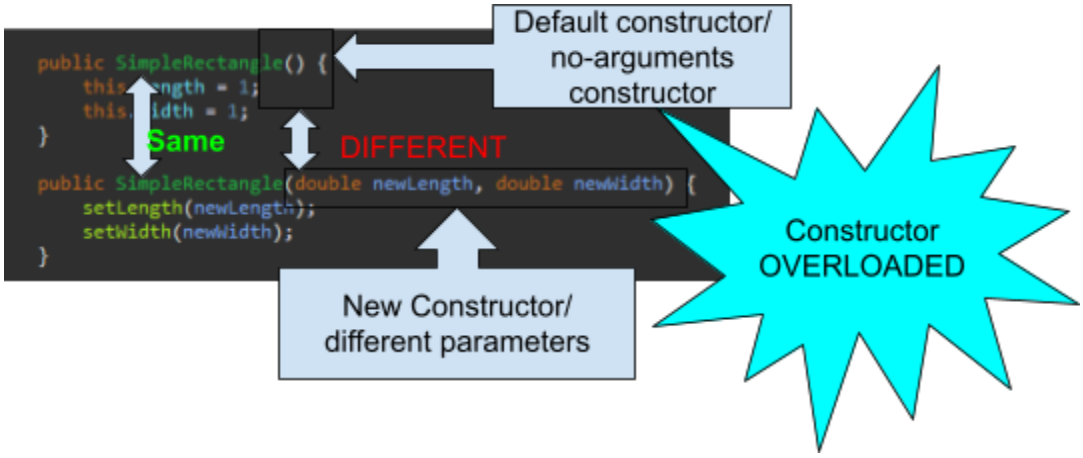
Default constructor:

- Has no parameters
- **Note**: if you don't explicitly make a constructor for a class, java will create one for you (no arguments, empty body, and does not affect instance variables)



How to overload a constructor:

- make more than one constructor, with the same name, and use different parameters.

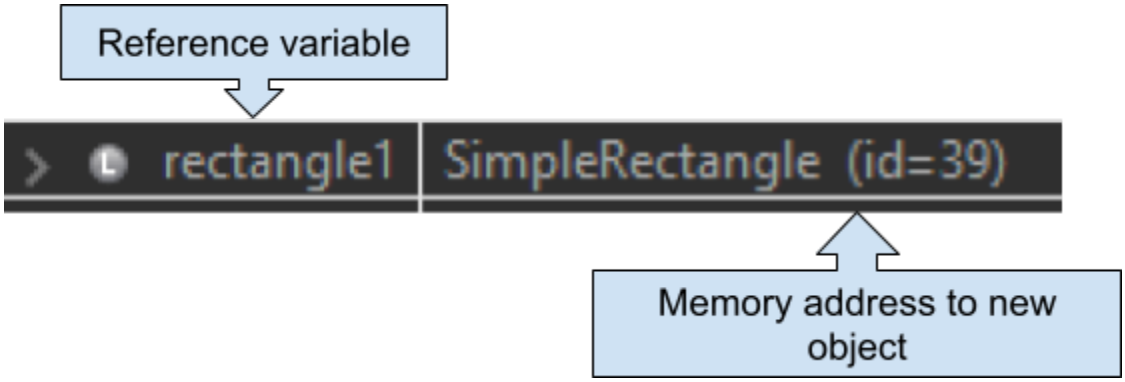


Invoking class constructors

How to invoke constructor:

- Use new operator. This invokes the class's constructor to create a new object
  - Allocates memory on the heap to store an object

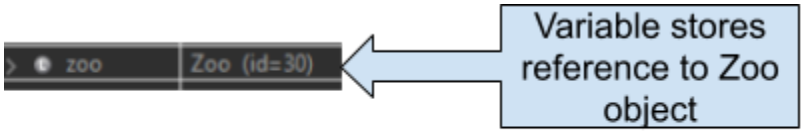
```
SimpleRectangle rectangle1 = new SimpleRectangle();
```

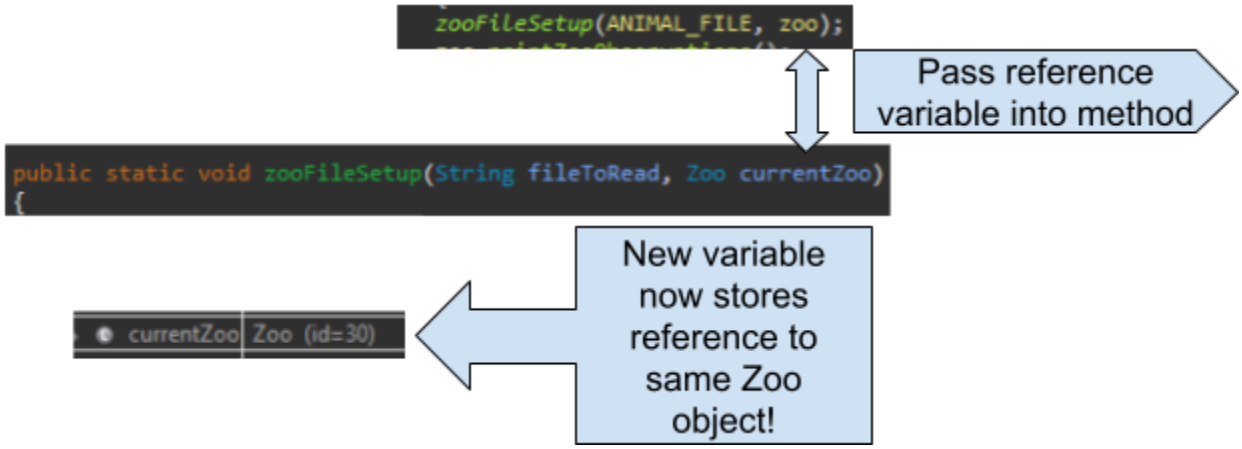


- The memory address for the newly created object is stored in the reference variable
- Reference variables are variables used to store memory addresses to objects on the heap

Passing by reference

- More information about passing by reference
- Objects are passed by reference
    - Instead of passing a copy of a value, we pass a reference of an object





- Making changes to an object, while in a method, affects the object everywhere else

Static vs nonstatic

- video about the difference between instance and static

- Information about instance and static methods

Instance variables and methods

- Instance variables and methods belong to a specific instance of a class
  - Not shared between objects of a class

When to use:

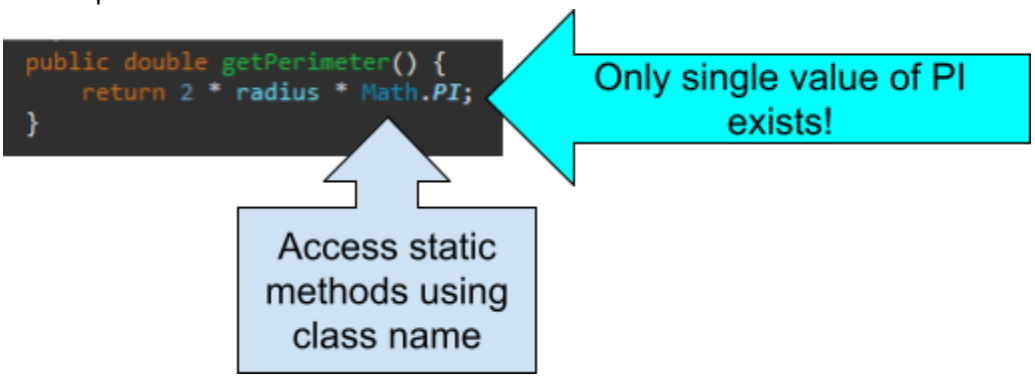
- Use if you need to access specific information about an object to use the method

Static variables and methods

- Belong to the entire class

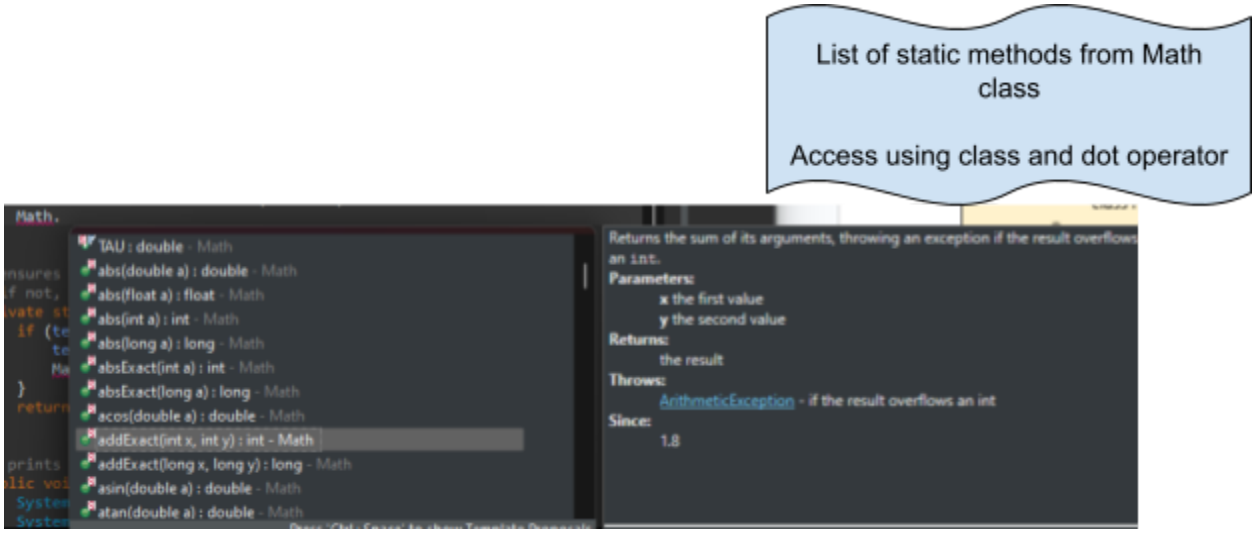
Static variables

- only ONE copy exists and is shared
- Can access using Class name or Object reference variable name, but better practice to use class name



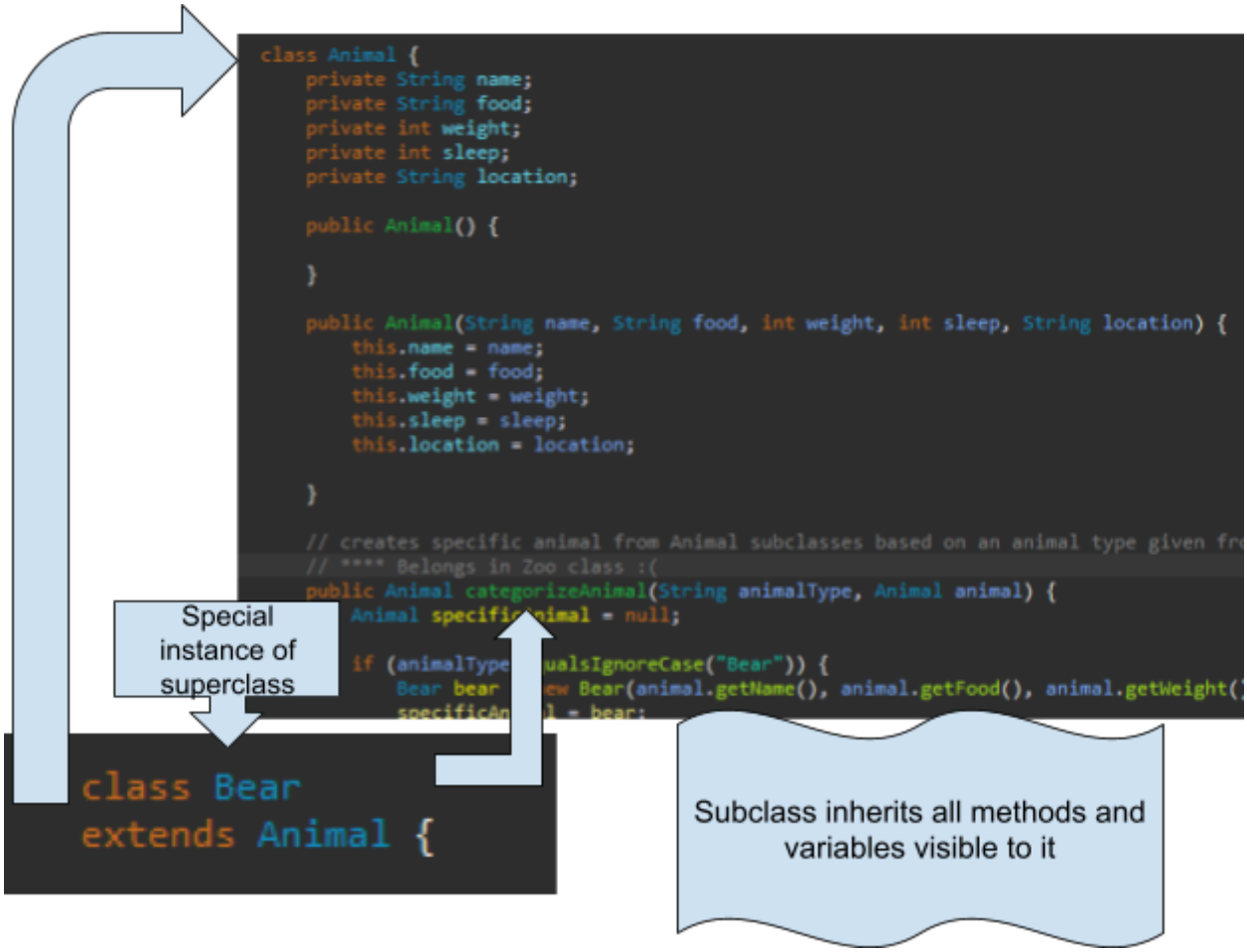
Static methods

- Use when it is not necessary to create an object to use it



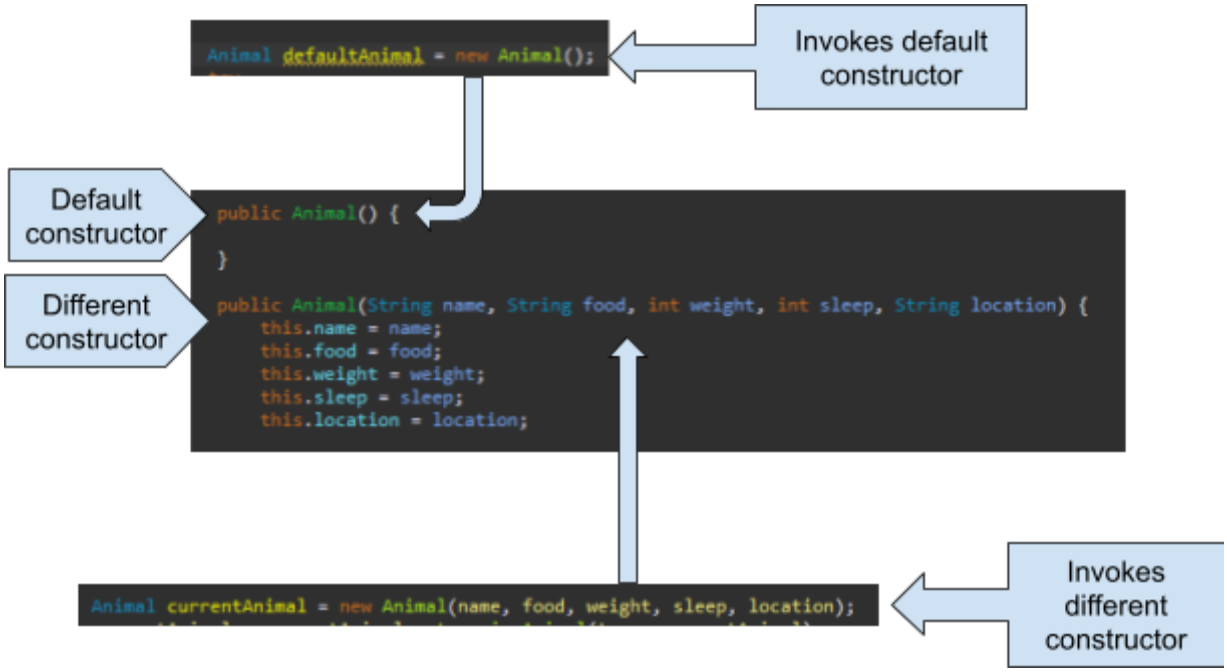
Inheritance

- Inheritance guarantees that a subclass has access to all methods and variables that are visible to it from its superclass
- A subclass object is a special instance of its superclass



SPolymorphism

- When you make methods with the same name but with different implementation
  - Method overloading
  - Method overriding



Polymorphism types:	Method Overloading	Method overriding
Has same name?	yes	yes
Has same parameters?	no	yes
Has same implementation?	no	no

Method overloading

- More information on method overloading
- Overload a method by keeping its method name the same, but changing the parameters and implementation

-

Method overriding

- More information on method overriding
- If parent class has a method that you'd like to change in a subclass, override it using @Override
    - Subclass object will use overridden method from its class rather than parent class
    -

```
class Bear
extends Animal {

    public Bear(String name, String food, int weight, int sle
        super(name, food, weight, sleep, location);
    }


    @Override
    public void eat() {
        System.out.println("Bear is eating " + getFood());
    }
```

Subclass methods > superclass method

JVM decides at runtime if overridden method executes

-

Dynamic binding

- More information on static versus dynamic binding
- There are things JVM determines at runtime instead of compile time, such as deciding which method to execute.
-

2D Array

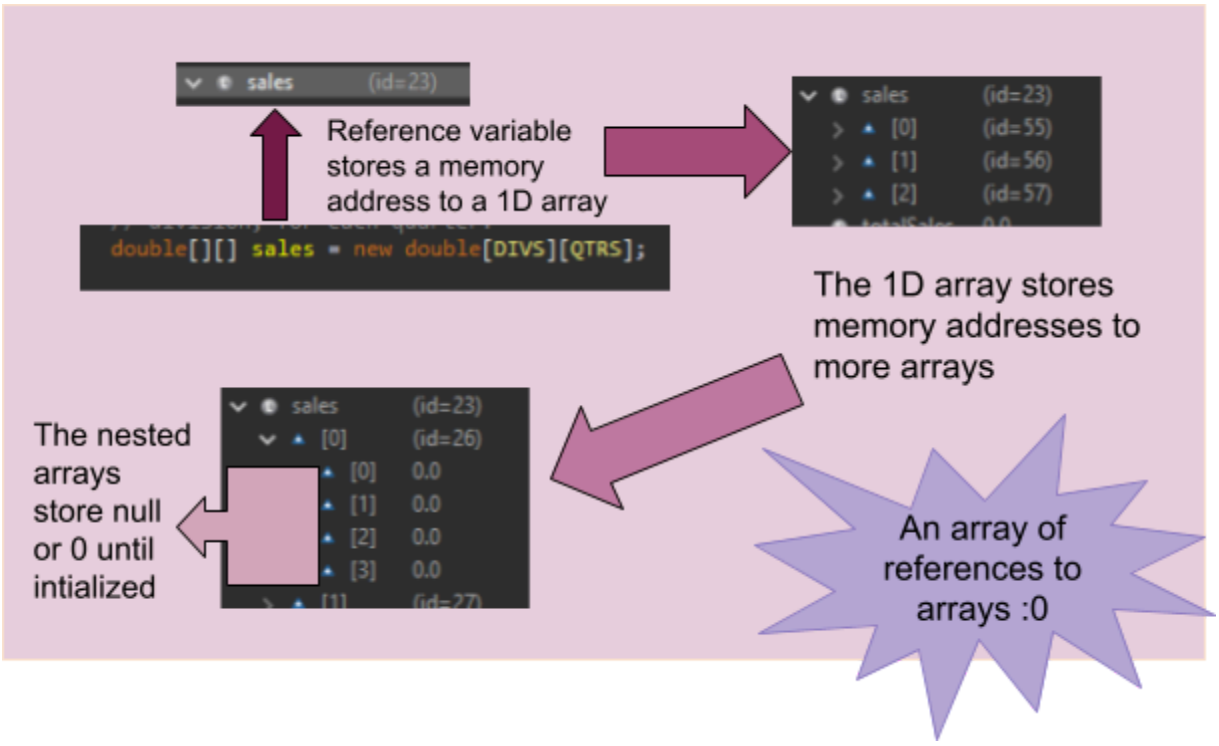
General information:

- A 2d array is an array of arrays

Comparison between 1D and 2D Arrays	
One-dimensional arrays	Two-dimensional arrays
Model linear collections of elements	Model a matrix or a table like a spreadsheet
double oneDArray[] = new double[SIZE];	double twoDArray[][] = new double[NUM_ROWS][NUM_COLUMNS];
oneDArray.length() gives number of indexes	twoDArray.length() gives number of rows twoDArray[0].length() gives number of columns

Using length()

- twoDArray.length() provides the number of rows in a 2D array
- twoDArray[0].length() provides the number of columns in a 2D array



How to initialize 2D array

With nested for loop

```
// first for loop iterates through each ROW of the 2D array
for (int row = 0; row < sales.length; row++)
{
	//nested for loop iterates through each COLUMN of the 2D array
	for (int column = 0; column < sales[row].length; column++)
	{
		//allows for user input to populate 2D array
		sales[row][column] = input.nextDouble();
	}
}
```

- Allows you to iterate through every position in the 2D array

With a list

```
int[][] testCases = {
		{ 4, 2, 7, 1, 5 }, // Regular case
		{}, // Empty array
		{ 5 }, // Single element
		{ 1, 2, 3, 4, 5 }, // Already sorted
		{ 9, 7, 5, 3, 1 }, // Reverse sorted
		{ 4, 2, 7, 2, 5 } // Array with duplicates
};
```

D Sorting 1D Arrays

- Link to animation for various sorting algorithms

Selection sort

- Link to animation
- Link to more information about Selection Sort
- Good for minimizing swaps
- too many comparisons for large arrays
- O(n^2) -> O(n^2)

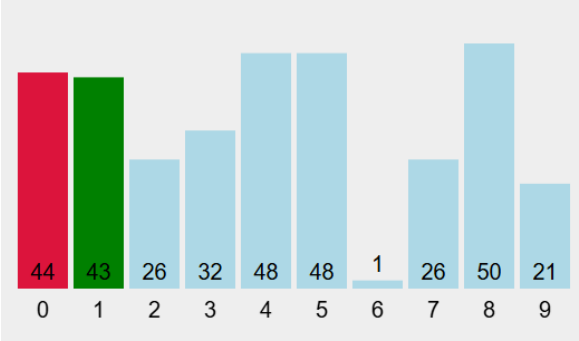
```java
// Selection Sort Algorithm
public static void selectionSort(int[] array) {
    int n = array.length;

    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;

        for (int j = i + 1; j < n; j++) {

            if (array[j] < array[minIndex]) {
                minIndex = j;
            }
        }

        int temp = array[minIndex];
        array[minIndex] = array[i];
        array[i] = temp;
    }
} //exit selectionSort method
```

Outer for loop sets point where we will swap with lowest value in array

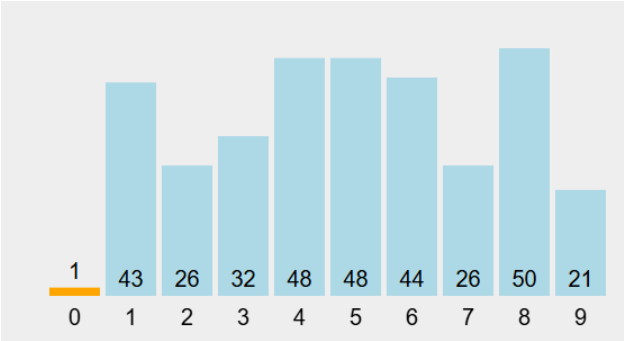
Nested for loop finds the smallest value in the array

Swaps current iteration's value with the smallest value

- *Links to more descriptions regarding code:*



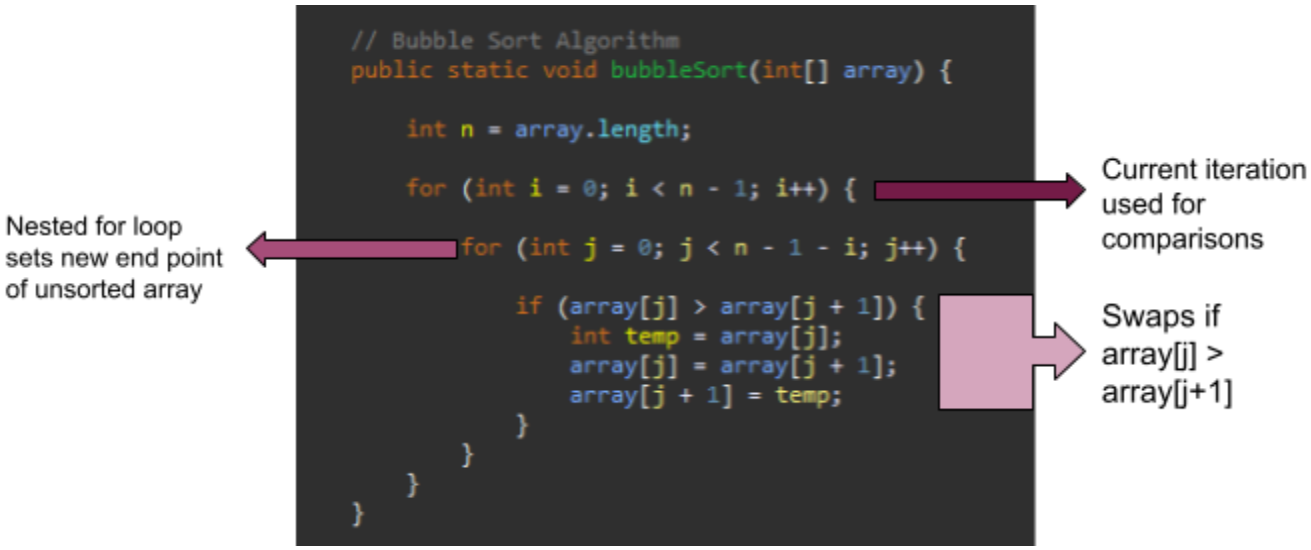
- Current index (in this case, value 44 at index 0) is compared to rest of array
- Smallest value, 1, is found at index 6



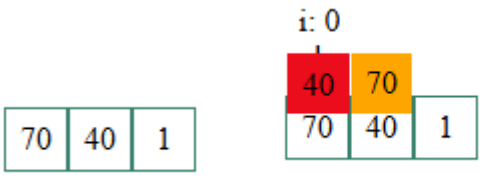
- After smallest value is found, it is swapped with current index, i (here it is 0)

Bubble sort

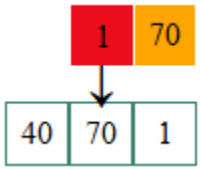
- Link to animation
- Link to more information about Bubble Sort
- best on small and nearly sorted arrays
- too slow for large arrays
- O(n) -> O(n^2)



- *Links to more descriptions regarding code:*



- In this case, index 0 (value 70) marks the current iteration we will use for comparisons
- Since index 1 (or rather, j+1) is greater than our current index 0 (j), we swap them



- After exiting inner for loop to return to outer for loop, the end of array is considered sorted so we no longer include it in our next iteration of inner for loop

Insertion sort

- Link to animation
- Link to more information about Insertion Sort
- Useful for when new elements are being added
- Good for small and nearly sorted arrays
- bad for large unsorted arrays

- O(n) -> O(n^2)

```java
// Insertion Sort Algorithm
public static void insertionSort(int[] array) {
    int n = array.length;

    for (int i = 1; i < n; i++) {

        int currentToSort = array[i];
        int j = i - 1;

        while (j >= 0 && array[j] > currentToSort) {
            array[j + 1] = array[j];
            j--;
        }

        array[j + 1] = currentToSort;
    }
}
```

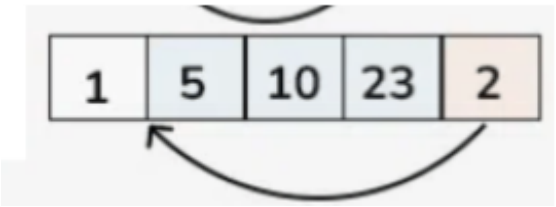
Starts at index 1, not 0

Compares value to values in sorted sublist

After exiting while loop, this inserts current value in correct position in sorted sublist

- *Links to more descriptions regarding code:*

- Index 0 is considered the start of the sorted sublist, so for loop iteration starts at index i = 1

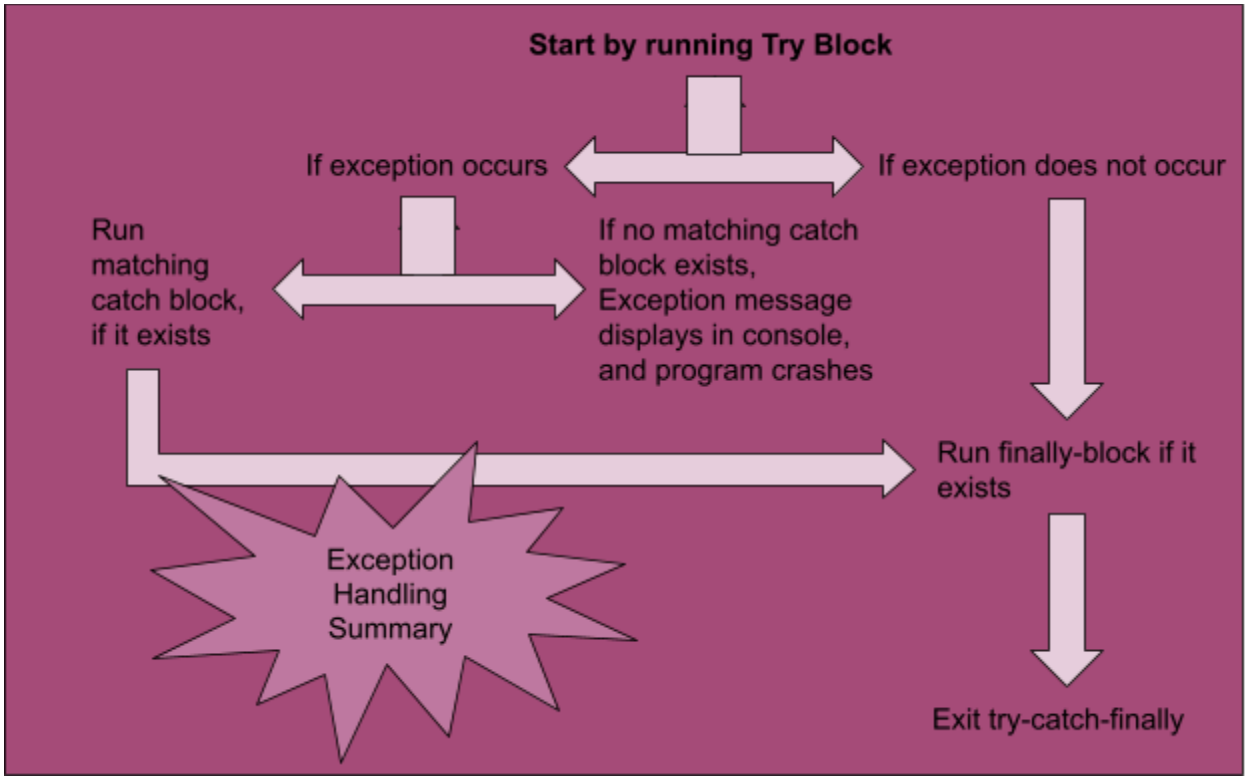


If the value of the current iteration's index is less than the number to the left of it, we continue moving left until we reach -1 or until we reach a number it is greater than

- Above, we continue moving left until we reach index 0 value 1
    - Here we stop and insert our value behind the index we exited the while loop at

Exception Handling

- We use try-catch-finally to handle exceptions in Java



Try-block

- Contains the code that may cause an exception
- If an exception occurs, the code exits the try block and immediately runs the matching Catch-block.

Catch-block

- Contains the code we might run if an exception occurs in the try-block
- Only the matching catch-block runs. If one doesn't exist, we get a normal exception message in the console and the program crashes

Finally-block

- Code at the end of a try-catch-finally that always runs whether an exception occurs or not
- Used to ensure we close files and release memory after finishing try-catch-finally

Search Algorithms

Linear Search

```
public static int linearSearch(int[] list, int key) {

    for (int i = 0; i < list.length; i++) {

      if (key == list[i])
        return i;
    }

    return -1;
  }
```

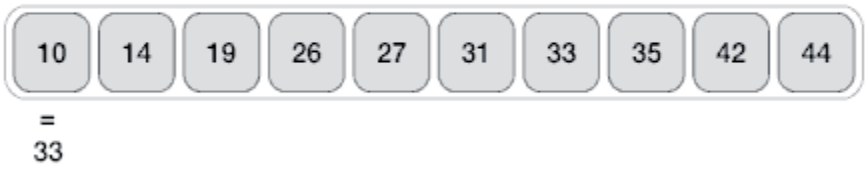
Iterate through entire array to search for key

If key is found, return index and exit code

If key isn't found, return -1

- *Links to more descriptions regarding code:*

Linear Search



- We Iterate through our array to search for a key
- If found, we return the index it was found at
- If index is not found, we instead return -1
- O(1) -> O(n)

Binary Search

- O(1) -> O(log n)

Arrays Class

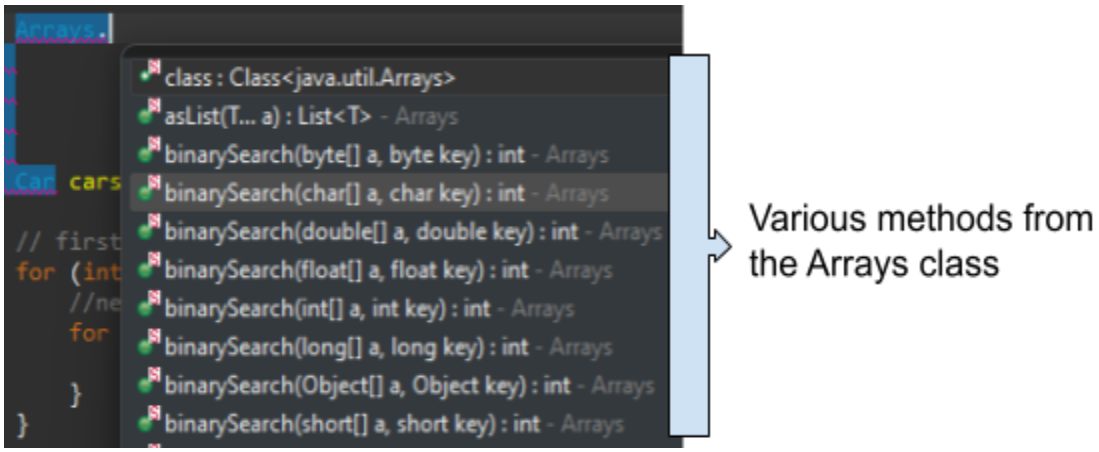
- You can use the Arrays class to use various methods including different sort methods

How to use Arrays class

- First we must import it:

```
import java.util.Scanner;
import java.util.Arrays;
```

- Since its a static class, we can access Arrays methods simply using Arrays and the dot operator



Algorithm Analysis

Performance

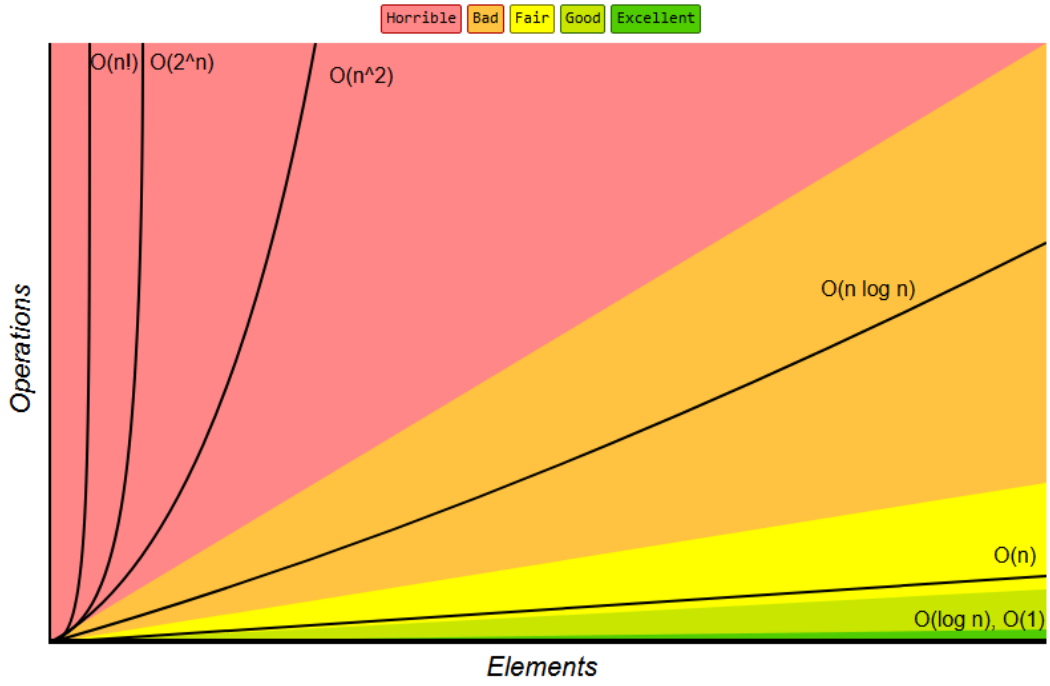
- Insertion sort: best for small and nearly sorted arrays
- Bubble sort: slowest overall. Lots of unnecessary swaps
- Selection sort: mid. Not as bad as bubble sort but not as fast as insertion sort

```
Testing with array size: 100
Selection Sort took 0.0938 ms
Insertion Sort took 0.0465 ms
Bubble Sort took 0.1725 ms
```

```
Testing with array size: 1000
Selection Sort took 3.9936 ms
Insertion Sort took 3.209 ms
Bubble Sort took 4.9494 ms
```

```
Testing with array size: 50000
Selection Sort took 788.0067 ms
Insertion Sort took 731.0938 ms
Bubble Sort took 4720.6907 ms
```

- Times are similar for smaller array sizes but the difference becomes significant for larger arrays sizes



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)

Cases

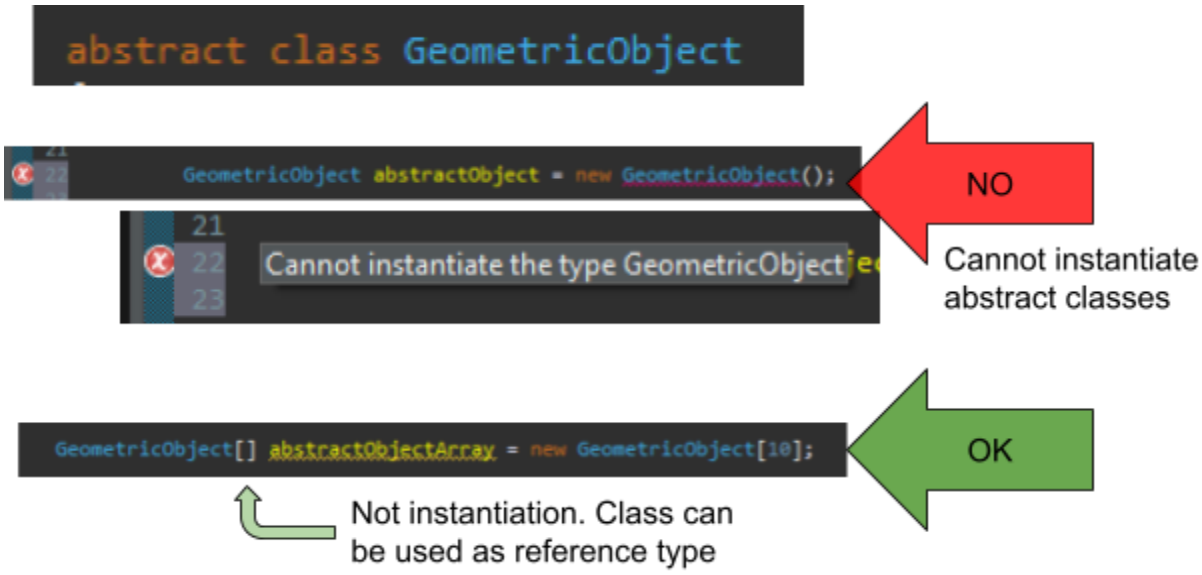
- Best case input: shortest execution time
- Average case input: typical execution time
- Worst case input: longest possible execution time

Big-O Notation

- This link talks more about the time growth rate of different algorithms
- O(n^2): selection sort, bubble sort, insertion sort, quick sort

Abstract Class

- Represents a generic object
- **Caution**: Cannot be instantiated
    - Attempting to instantiate an abstract class causes an error
    - CAN be used as a reference variable type

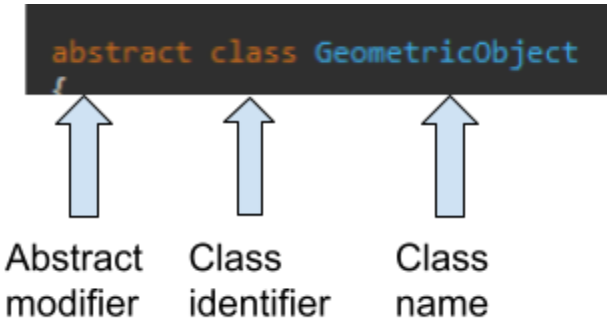


- Must be extended to be of any use
- Abstract classes define behaviors for its subclasses.
    - Subclasses must implement behaviours

Abstract Class or Concrete Class?		
	Answered: “No”	Answered: “Yes”
Is the class specific enough to make objects?	abstract	concrete
Do objects of this type make sense?	abstract	concrete
Some behaviors stop making sense unless implemented by a subclass	concrete	abstract
Do we need the class for inheritance and/or polymorphism?	concrete	abstract
Can we use it as a generic reference type?	NA	concrete/abstract

How to define abstract class

- Class Name
    - Use abstract modifier
    - Use class identifier as with concrete classes.

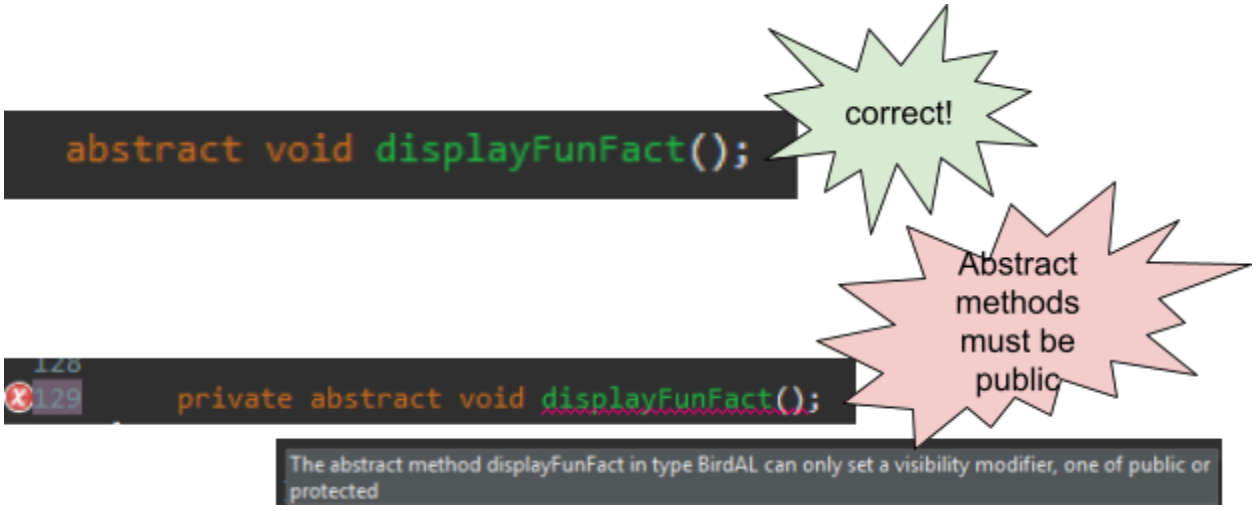


Rules for abstract classes

- If one or more methods is defined abstract, the class must be defined abstract
- A class can be defined abstract even though no methods are defined abstract
    - This prevents the user from creating objects from a “too generic” sounding
- A subclass of an abstract class must override all abstract methods otherwise it must be defined as an abstract class

WARNINGS

- Abstract classes CANNOT be defined final nor static
- Abstract methods CANNOT be defined final nor static



- *Why?*
    - final modifier doesn’t allow us to override method and method must be overridden if abstract

- static modifier indicates that we do not need to create an object to use the method, but we need an object to override in the first place!

Why use abstract classes?

- We need them for polymorphism. Although it cannot be used to instantiate an object, it can be used to group different classes together under a parent class where they all share certain behaviors.

Interface

- A *pure* abstract class
    - All methods in an interface are abstract
    - All data members are constants. NO instance variables! All data members are automatically defined public static final
    - All methods are inherently public and abstract so those keywords are unnecessary
        - Cannot be private nor protected

- **Why use?**
    - Provides full abstraction. Separates the “what to do” from “the how to do.”
    - Allows you to write more flexible code

Defining Abstract class vs Interface		
Keyword used	abstract	interface
Data members	All types of data allowed	Can only be constant. (public static final, only)
Constructor?	yes	no
Can have concrete methods?	yes	no
How to define subclass using it?	Keyword: extends	Keyword: implements
Can be used as a reference data type?	yes	yes

Defining Abstract class vs Interface		
Methods can be concrete?	yes	no

ArrayList

Array vs ArrayList		
	Array	ArrayList
Part of Java Collections Framework (JCF)	No	Yes
Size	static Cannot be changed once initialized	dynamic Can add() and delete() elements as needed
Implement interfaces?	cannot	Can implement several interfaces such as LIst, Set, Map
Store elements contiguously?	Yes	Yes
Declaration	dataType referenceVariable;	ArrayList<ClassType> referenceVariable
Initialization	referenceVariable = new dataType[FIXED_SIZE];	referenceVariable = new ArrayList<>();

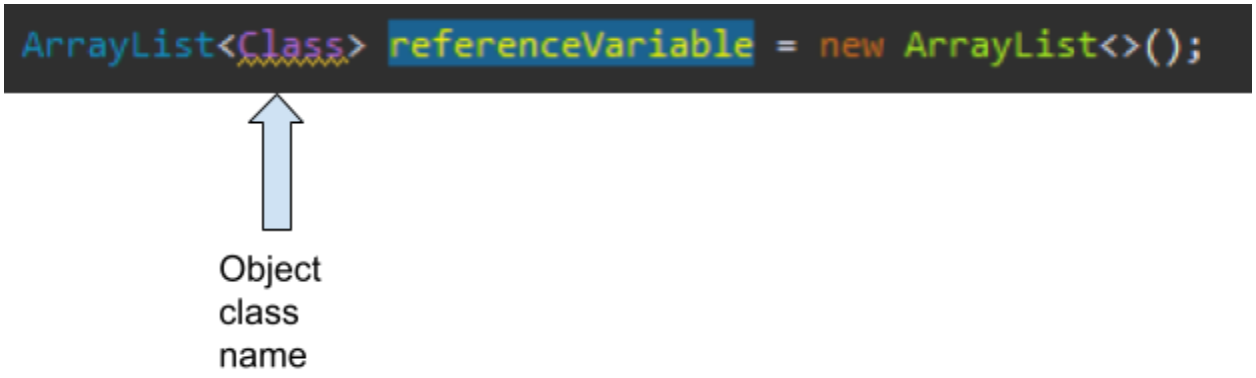
How to use ArrayList

- Import java.util.ArrayList

```
import java.util.ArrayList;
```

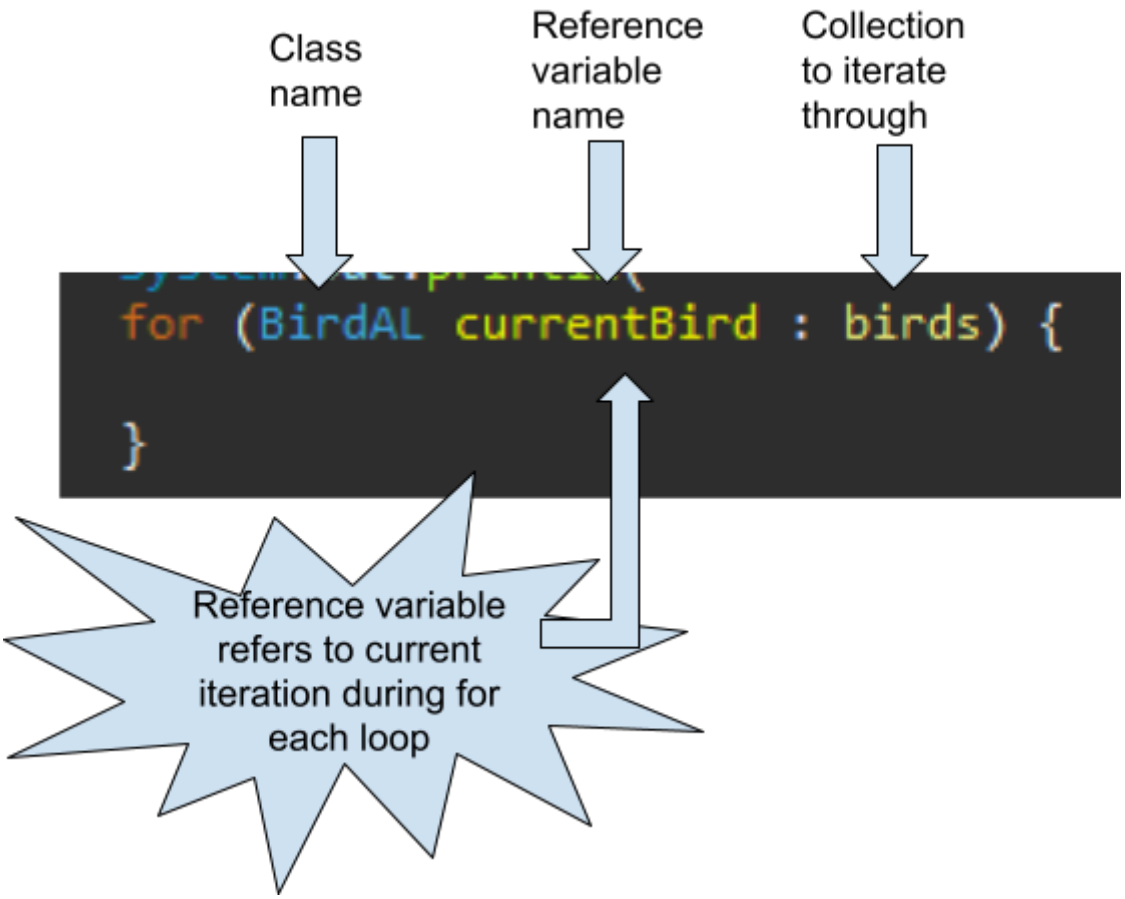
- Declaration and initialization
    - ArrayList<String> stringList = new ArrayList<>();

- Because its a generic class, use angle brackets



For-each loop

- An enhanced for-loop used to iterate through collections, like ArrayList
- Great for displaying elements in ArrayList in normal order
    - Bad if you are iterating through to modify
    - **If you need anything funky, use traditional for-loop**



Casting with inheritance

Implicit casting (upcasting)

- Happens automatically when we assign a subclass object to a superclass reference (smaller to bigger)

Explicit casting (downcasting)

- Must be explicitly done carefully
    - If done incorrectly, may throw ClassCastException
- Why do we use?
    - We cannot access subclass specific methods if they are not available in the reference variable data type class

Basic ArrayList Operations	
Size method	arrayList.size();
iteration	Method 1: For-each loop Method 2: For loop
Accessing from	get()
Removing from	remove()

Stacks and Queues

Linear data structures: Stack vs Queue		
	Stack	Queue
Where are insertions made?	Tail of stack push()	Tail of queue enqueue()
Where are deletions made?	Tail of stack (LIFO) pop()	Head of queue (FIFO) dequeue()

Linear data structures: Stack vs Queue		
What does peek() do?	Returns the last element	Returns the front element
Example uses?	undo/redo functions back/forward buttons on browser	Customer service lines Task scheduling orders
Exception thrown?	NoSuchElementException	IllegalStateException

Basic Stack Operations using an ArrayList `ArrayList<Integer> intList = new ArrayList<>();`		
Stack operation	What does it do?	ArrayList equivalent
push()	Adds element to the top of the stack	add() adds to the end of ArrayList `intList.add(10);`
pop()	Removes from the top of the stack	Remove() and size()-1 allows us to remove the last element on the ArrayList `intList.remove(intList.size()-1);`
peek()	Returns the top of the stack (does not remove)	This will return the element at the end of the ArrayList but will not delete it. `intList.get(intList.size()-1);`

Basic Queue Operations using an ArrayList		
Queue operations	What does it do?	ArrayList equivalent
enqueue()	Adds to the end of the queue	add() adds to the end of the ArrayList

Basic Queue Operations using an ArrayList		
		`intList.add(10);`
dequeue()	Removes from the front of the queue	Removes the first element on the ArrayList `intList.remove(0);`
peek()	Returns the front of the queue (does not remove)	Returns the first element on the ArrayList `intList.get(0);`

Linkedlist:
To use, import java.util.LinkedList

```
    /**
     * this method inserts node into linked list in numerical order
     *
     * @param number: used to initialize new node we will insert into linked list
     */
public void insertNode(int number) {
    NodeFix newNode = new NodeFix(number);
    NodeFix current = head;
    NodeFix previous = null;
    while (current != null && current.data < number) {
        previous = current;
        current = current.next;
    }


    // checks if we are are at start of linked list
    // previous == null indicates that current is the head
    if (previous == null) {


          // inserts the node we created at the start of the linked list

        newNode.next = head;// new head points to original head
        head = newNode; //we set head to our new node

    // for inserting anywhere else (including tail)
    } else {
          newNode.next = current; //newn=Node points to what previous was pointed at
                                  //newNode.next = previous.next;
```

```
            previous.next = newNode; //previous now points to our new node
        }
    }
```
]

Abstract Data Type

- Data type that isn't pre-defined in the programming language.
- The details of its implementation should be hidden.
- A ***collection*** is an abstract data type.

Data Structure

- Underlying programming constructs and techniques used to implement a ***collection.***
- Arrays, array lists, stacks, queues, and linked lists are considered data structures.

Dynamic Data Structures

- A data structure that can grow and shrink during execution time.

Dynamic Memory Allocation

- Obtaining and releasing memory during execution time.

Object Reference

- Variable that stores the address of an object.

Self-Referential Class

- When a class contains a member that is a reference (pointer) to its same class type.

Linked Lists

- Linear collection of self-referential structures.
- Insertions and deletions are made anywhere in the list.

```java
//singly linked list method.
 public void insertNode(int number) {
      NodeFix newNode = new NodeFix(number);
      NodeFix current = head;
      NodeFix previous = null;
      while (current != null && current.data < number) {
          previous = current;
          current = current.next;
      }


      // checks if we are are at start of linked list
      // previous == null indicates that current is the head
      if (previous == null) {


          // inserts the node we created at the start of the linked list
          newNode.next = head;// new head points to original head
          head = newNode; //we set head to our new node

      // for inserting anywhere else (including tail)
      } else {
      newNode.next = current; //newn=Node points to what previous was pointed at
                                            //newNode.next = previous.next;
          previous.next = newNode; //previous now points to our new node
      }
  }
```

```java
      //doubly linked list method
      public void deleteNode(int data)
      {
              // Implement this
              System.out.println("Implement delete node:");
              NodeL18 current = head;
          // traverses until we find the node or go through the whole list
          while (current != null && current.data != data) {
              current = current.next;
          }

          //handles if the node to delete is the head
          if (current == head) {
              head = current.next;
              if (head != null) {
                  head.prev = null;
              }
          }

          else {
              current.prev.next = current.next;
              if (current.next != null) {
                  current.next.prev = current.prev;
              }
          }
      }
```

```java
	//doubly linked list method
	public void insertAtHead(int data)
	{
		System.out.println("Implement Insert at the Head:");

	    NodeL18 newNode = new NodeL18(data);
	    //handles if list is empty
	    if (head == null) {
	        newNode.next = null; // or newNode.next = head. up to u. still points to null


	    // handles if list is NOT empty
	    } else {


	       // handles links between newNode and old head
	        newNode.next = head;
	        head.prev = newNode;

	    }
	    newNode.prev = null; // since newNode is new head, its prev points to null
	    head = newNode;
	}
```

```java
	//doubly linked list method
	// Insert at the end (already implemented)
	public void insertAtEnd(int data)
	{
		NodeL18 newNode = new NodeL18(data);
		if (head == null) // handles if list is empty.
		{
			head = newNode; //no point in traversing. just insert
		} else
		{
			NodeL18 temp = head;
			while (temp.next != null) //traverses through list until we reach tail
			{
				temp = temp.next;
			}
			temp.next = newNode; //links tail to newNode
			newNode.prev = temp; // Backward link
		}
	}
```