WEEK ELEVEN

Acknowledgements: Slides created based off material provided by Dr. Travis Doom

STATIC KEYWORD

Static Method

- Can be executed without a class instance (object)
- Not dependent on the state of an object
- CAN'T use instance variables (directly)
- Can use static and local variables
- Can't use non-static methods of the class

```
int num = Math.abs(-5);
```

Non-Static Method

- Must be called via a class instance (object)
- Dependent on the state of the object
- Can use instance and local variables
- Can call non-static methods of the class

```
ArrayList<Integer> nums = new
ArrayList<>();

nums.add(5);
```

STATIC KEYWORD

Static Variable

- Variable has one memory location shared by all instances of the class (objects)
- One value per class, not per object
- Can be accessed without an object

Non-Static Variable

- Variable has the same number of memory locations as there are instances of the class
- One value per object, not class
- Must be accessed via an object

STATIC VARIABLE EXAMPLE

```
public class Pixel {
	public static int numPixels = 0;
	public int x;
	public int y;
	…
}
```

```
Pixel p1 = new Pixel();
p1.x = 5;
p1.y = 5;
Pixel.numPixels++; // numPixels = 1
Pixel p2 = new Pixel();
p2.x = 35;
p2.y = 74;
p2.numPixels++; // numPixels = 2
```

SOFTWARE DEVELOPMENT

- Analysis
- Design
- Implementation
- Testing
- Maintenance

ANALYSIS

- Description of the desired system/application
  - Tasks it performs
  - Provided data
  - Specifications
- Could involve working with a non-programmer
  - Non-technical language
  - General descriptions

DESIGN

- Decomposition
  - Break requirements down into subsystems/subcomponents
  - Utilize objects, methods to simplify problem
- Pseudocode
  - Describe task in general programming terms
  - Identify useful programming structures (conditionals, loops, methods)
  - Syntax does not need to be followed
  - Verify correctness of logic
- Diagrams
  - UML, flowcharts, etc.

IMPLEMENTATION

- Select language
- Determine necessary packages/libraries, input data
- Utilize good decomposition and structure
- Provide thorough documentation and clean style
- Often need to redesign portions

TESTING

- Critical step
- Perform tests with sample data, inputs, user interactions, etc.
- May test components individually before testing the entire design
- Find and squash bugs (errors in code)
- Debugger is useful for determining cause of issues

MAINTENANCE

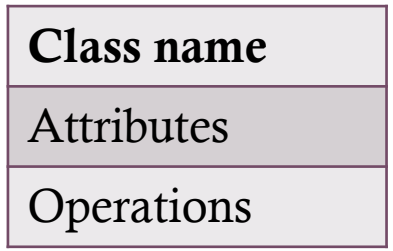
- Ensure program continues to work
  - Account for updates to dependencies (packages/libraries)
  - Account for hardware upgrades or wider system compatibility
- Continue to document changes
- Built-in automated tests to avoid new errors
- Often, software development cycle is not linear

OBJECT ORIENTED DESIGN

- Look for nouns in problem description/requirements
  - Potential classes
- Refine the list
  - Look for nouns that may mean the same thing or be represented in the same way
  - Check for nouns that are to broad or not necessary for the problem
  - Determine if each noun represents a class or instance of the class (object)
  - Check for nouns that could be stored as a variable in a class
- For each class
  - Determine what it knows (instance variables)
  - Determine what it does (methods)

UML DIAGRAMS

- Basic structure
  - Class name
  - Attributes (instance variables)
  - Operations (methods)
- Access modifiers are also specified
  - + : public
  - - : private

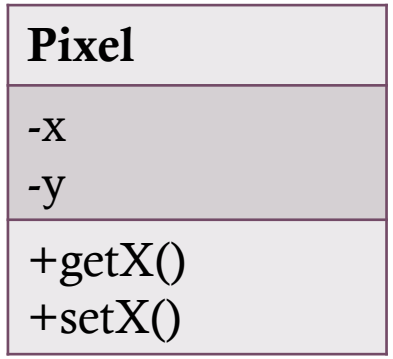


UML DIAGRAMS CONTINUED

```
public class Pixel {
	private int x = 0;
	private int y = 0;

	public int getX() {
		return x; }

	public void setX(int newX) {
		x = newX; }
}
```



UML DIAGRAMS CONTINUED

- Signatures can also be specified
  - Data type for attributes
  - Parameters for methods
  - Return types for methods

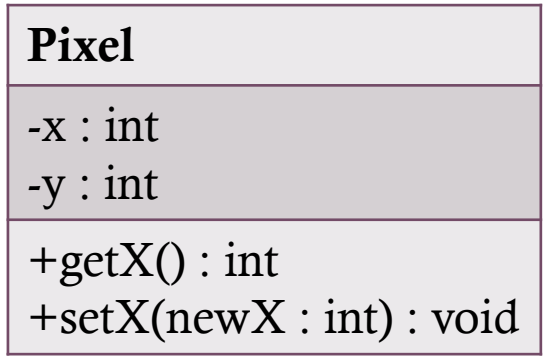
Class name
Attributes : data type
Operation(name : data type) Operation : return type

UML DIAGRAMS CONTINUED

```
public class Pixel {
	private int x = 0;
	private int y = 0;

	public int getX() {
		return x; }

	public void setX(int newX) {
		x = newX; }
}
```



GOOD CONCEPTS TO FOLLOW

- Abstraction: ability to view a complex operation in a simplified form
  - Good classes are abstract representations of objects in the problem domain
- Method Cohesion: degree to which a method implements a single function
  - Methods should execute a well-specified task
  - No surprise side-effects
  - Reusable
- Precondition/postcondition enforcement (defensive programming)
  - Enforce preconditions by verifying input is as expected/necessitated
  - Maintain postcondition consistency

DOCUMENTATION GUIDELINES

- Each class header:
  - Short description of the general purpose of the class
- Each method header:
  - Brief description of method function, parameters, return type
  - Any preconditions/postconditions
- Internal documentation:
  - Description of vague/potentially confusing variables
  - Comment per block of significant code
  - Comment for any strange/unusual operations