METRO STATE UNIVERSITY

Data Types

Primitive Data Types

Classes & Abstract Data Types (ADT)

METRO STATE UNIVERSITY

Primitive Data Types

Primitive Data Types can be represented with 0's and 1's.

Integers

- byte
- short
- int
- long

Decimal Numbers

- float
- double

Letters

- char – ' ' not " "

Boolean values

- boolean – true, false

Primitive Data Types can be represented with 0's and 1's.

Integers

- byte
- short
- int
- long

Decimal Numbers

- float
- double

Letters

- char – ' ' not " "

Boolean values

- boolean – true, false

Each **primitive** data type also has

✓ A set of specific values it can have

✓ Predefined operations

Primitive Data Types can be represented with 0's and 1's.

Integers

- byte
- short
- int
- long

Decimal Numbers

- float
- double

Letters

- char – ' ' not " "

ues

– true, false

This works well for computers but not always in real life

ata type also has

- ✓ A set of specific values it can have
- ✓ Predefined operations

Can we use computers to model real-world entities / systems?

- How to we represent things other than primitive data?
- Can I represent / model a car a person? A house?

METRO STATE UNIVERSITY

Abstract Data Types - Classes

METRO STATE UNIVERSITY

What if you want to represent something that is not a primitive data type?

- We follow the similar approach as for defining primitive data types
  - Find a way to express ***the states*** or store ***data*** about the new data type
  - Define the ***set of operations*** that can be done on the new data type

- The non-primitive data type is known as ***Abstract Data Type*** (ADT) or ***Object***(from a class in Java)

What if you want to represent something that is not a primitive data type?

- We follow the similar approach as for defining primitive data types
  - Find a way to express ***the states*** or store ***data*** about the new data type
  - Define the ***set of operations*** that can be done on the new data type

- The non-primitive data type is known as ***Abstract Data Type*** (ADT) or ***Object***(from a class in Java)

Defining an Abstract Data Type

- Find a way to express the states or store data about the new data type
- Define the set of operations that can be done on the new data type
  - We use ***methods***
    - Initialization
    - Obtain the values of the various fields
    - Modify the fields
    - Compare different instances of the same object

Defining an Abstract Data Type

- Find a way to express the states or store data about the new data type
- Define the set of operations that can be done on the new data type
  - We use ***methods***
    - Initialization -- *Constructors*
    - Obtain the values of the various fields -- *Getters*
    - Modify the fields -- *Setters*
    - Compare different instances of the same object

Abstract Data Type (ADT)

- An ADT gives the description of a given data structures.

- An ADT is defined by:
  - Data organization.
  - Operations (methods) that operates that data structure
  - Each operations is defined by:
    - description of what the operation does.
    - number and data type of input parameters.
    - type of output.

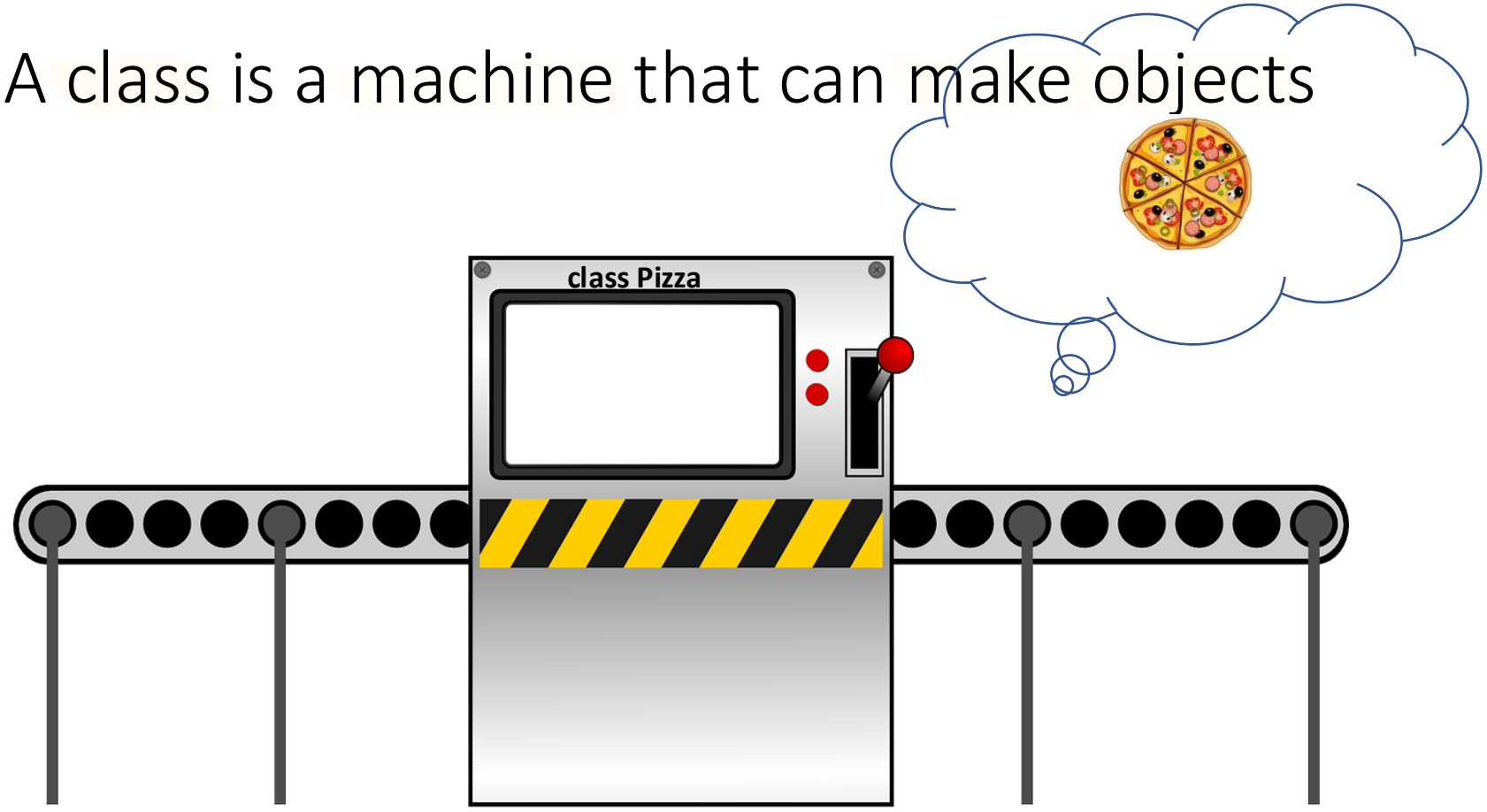
- An ADT is implementation-independent:
  - It just specifies what each operation does but not how it does it

Implementing Abstract Data Types

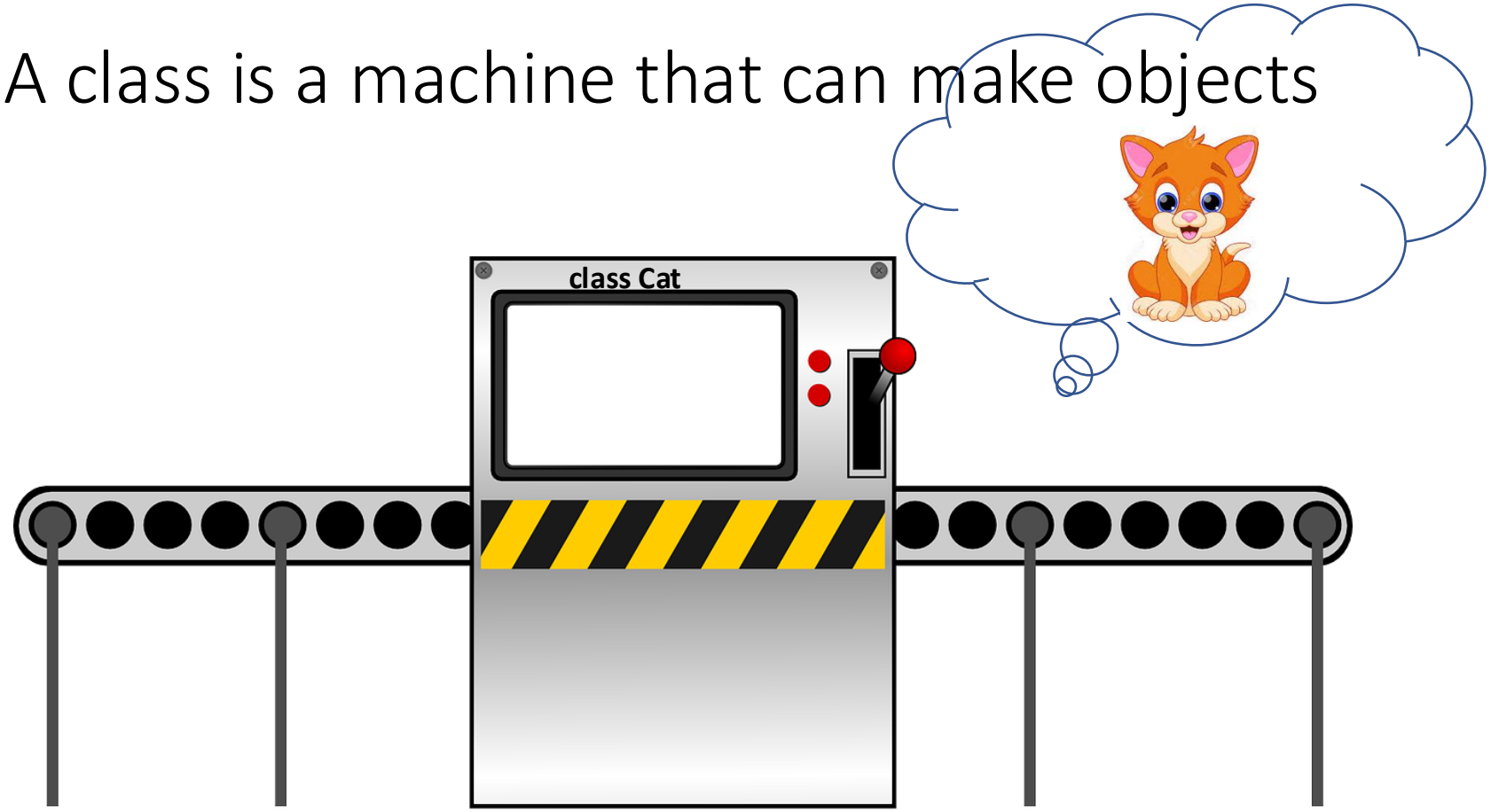
- Abstract data types are defined to be implementation-independent.
- An ADT can be implemented in different ways using different programming languages.
- The implementation must respect the interface that is defined in the ADT definition.
- The implementation should not make any assumptions about how the data structure will be used.

Another way to look at it…

A class is a machine that can make objects



A class is a machine that can make objects



Let’s make a class in Java

public class _____{

}

*Cat.java

```
1
2  public class Cat {
3
4  }
5
```

*Pizza.java

```
1
2  public class Pizza {
3
4  }
5
```

4 - Try it!

- Create a project called Practice

- Create a class to represent the following:
  - Pizza
  - Student
  - Car
  - Television

Customization

We can add the ability to customize

Pizza
size number of toppings gluten free or not

Cat
name age lives outside or not

What questions might you ask about a pizza? A cat?

5-Try it!

Choose another object and come up with three fields. For now, choose fields that are able to be represented as a letter, number, or true/false (primitive data types). Please format in pseudo-UML.

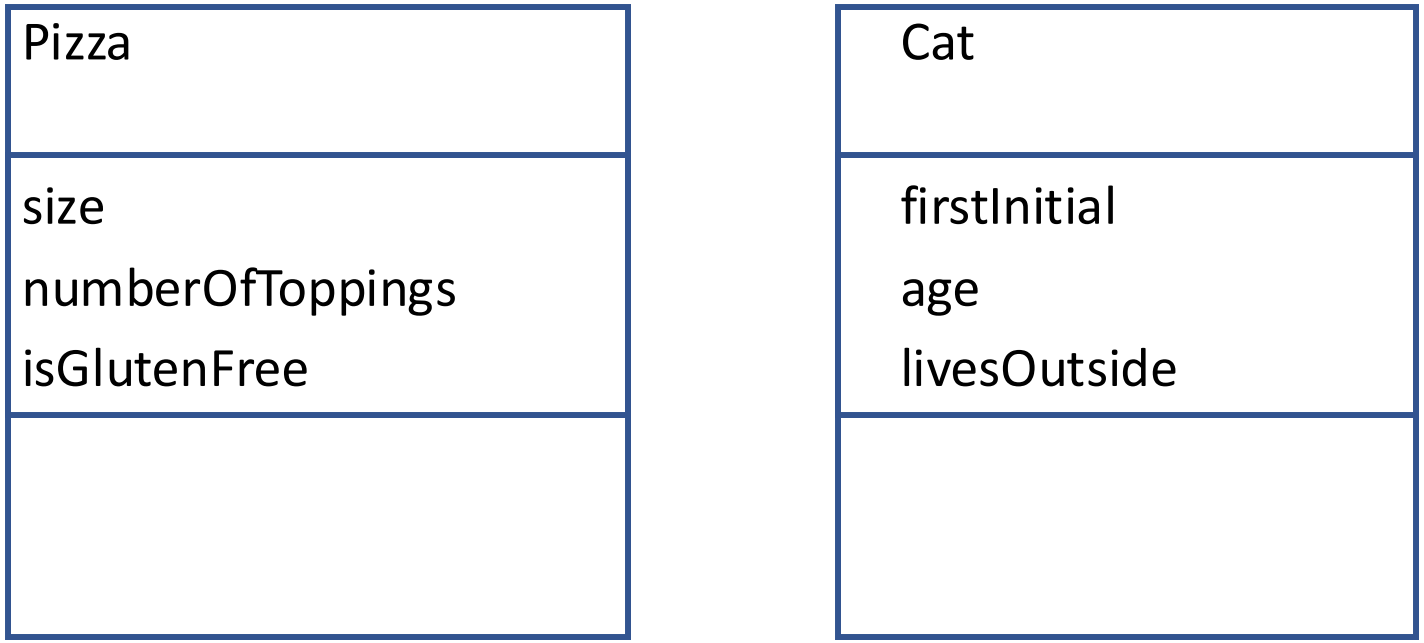
****Must describe an individual thing, not a set. For example: shoe size vs number of shoes

- Shoe size is a description of one pair of shoes
- Number of shoes is a description of a collection of shoes

Those descriptions can be turned into variable names.

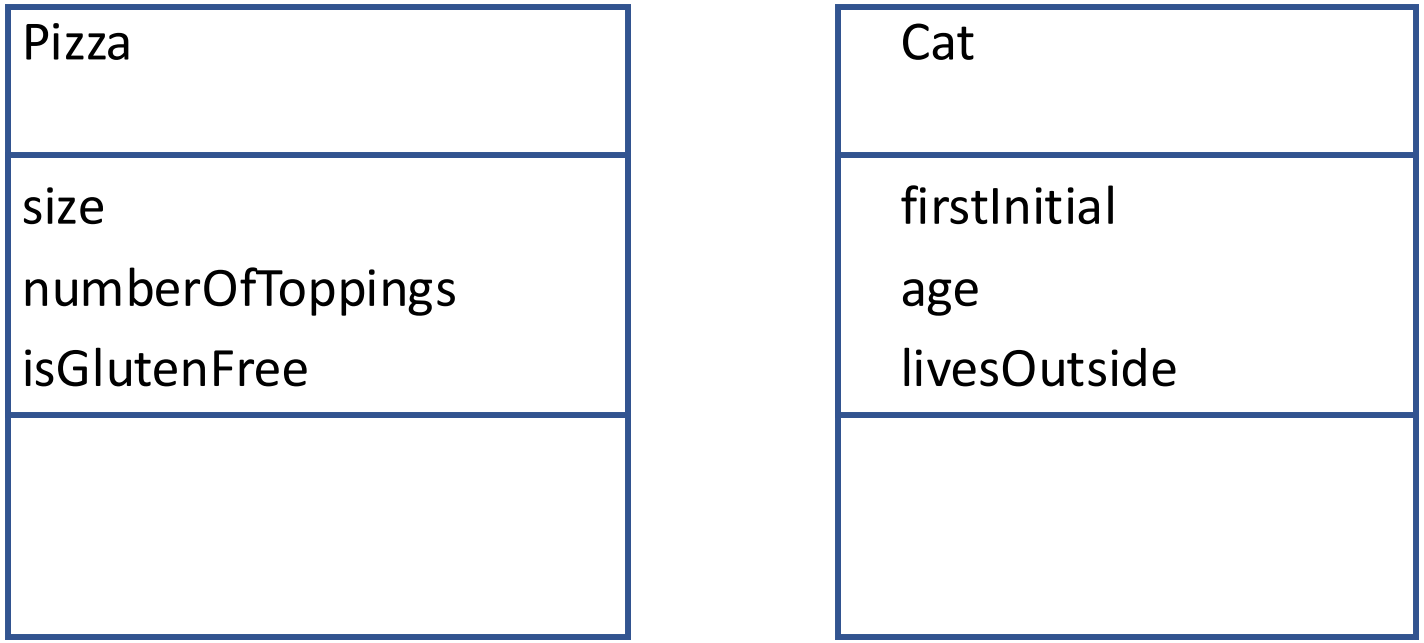
- Naming conventions are almost the same as Python
  - Letters and numbers
    - Can use $ and _ (but don't)
  - Start with letter
  - Use full words (size instead of s)
  - No keywords or reserved words
  - Capitalize constants
- Different:
  - numberOfToppings instead of number_of_toppings
  - Can use _ when multiple word constant NUM_TOPPINGS

Turn into variable names



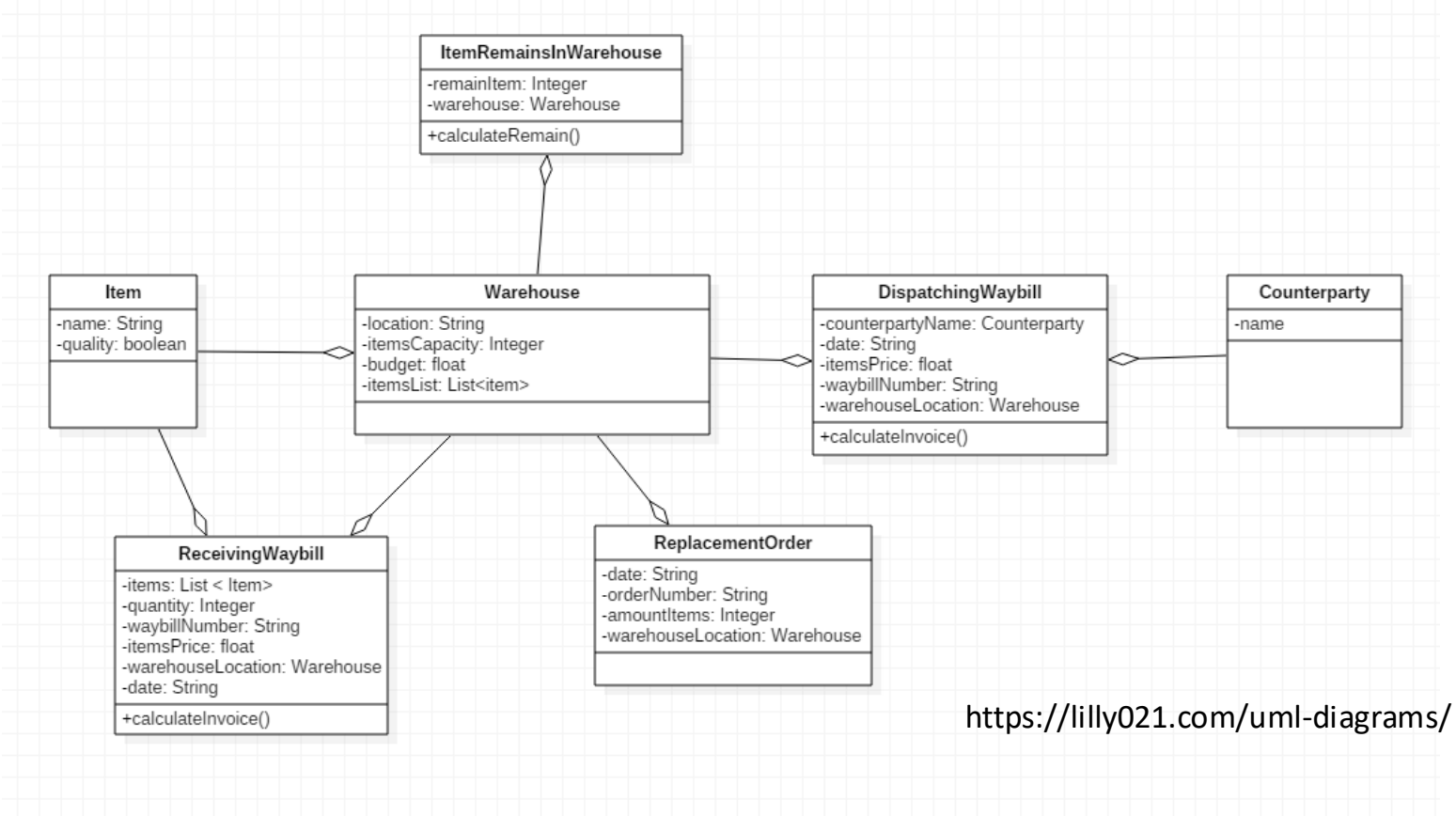
** Officially “Fields” – Specifically Instance Variables

Turn into variable names



Why the weird boxes?

UML Diagram (Unified Modeling Language)



https://lilly021.com/uml-diagrams/

UML Diagrams

UML Notations

- Access Modifiers:
  - public is represented by `+`
  - private is represented by `-`

- Variables:

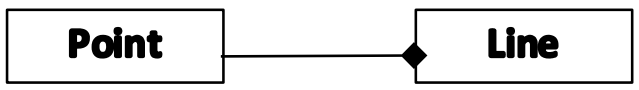
```
<access_modifier> <variable_name>: <data_type>
-x : double
```

- Methods (note: `static` methods are underlined)
  - `+setX(x:double):void`
  - `+midPoint(p1:Point, p2:Point):Point`

`has-a` Relationship

- A class has an instance variable of another class
  - Examples:
    - A `Line` has two variables of type `Point`
    - A `Course` has an array of `Student`

- Has-a relationships must be shown in UML diagram



Good programming practices

- You must follow the following in all your programming projects:
  - Program is readable.
  - Consistent and correct indentation.
  - Meaningful variable names.
  - Using Java variable-naming conventions.
  - Use `Javadocs` comments (see Appendix H).

Java Naming Conventions

- Variables and method names:
  - Starts with a lower case letter
  - If the name consists of several words, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name.
  - Examples:
    - `radius`
    - `area`
    - `computeArea`

- Class names begin with upper case:
  - `Line`

Javadocs Comments

- Include comments on the class as a whole including:
  - Description of the program
  - Your name (using `@author`)

- For each method include:
  - Concise description of what the method does.
  - Description of input parameters (use `@param`)
  - Returned value (use `@returns`)
  - Exceptions what are thrown (use `@throws`)

Javadocs: Appendix H

- Write the following:
  - `@author`
  - For each method:
    - description
    - `@param` – input variable name
    - `@return`

- From Eclipse, `Project -> Generate javadoc`

Declaring variables

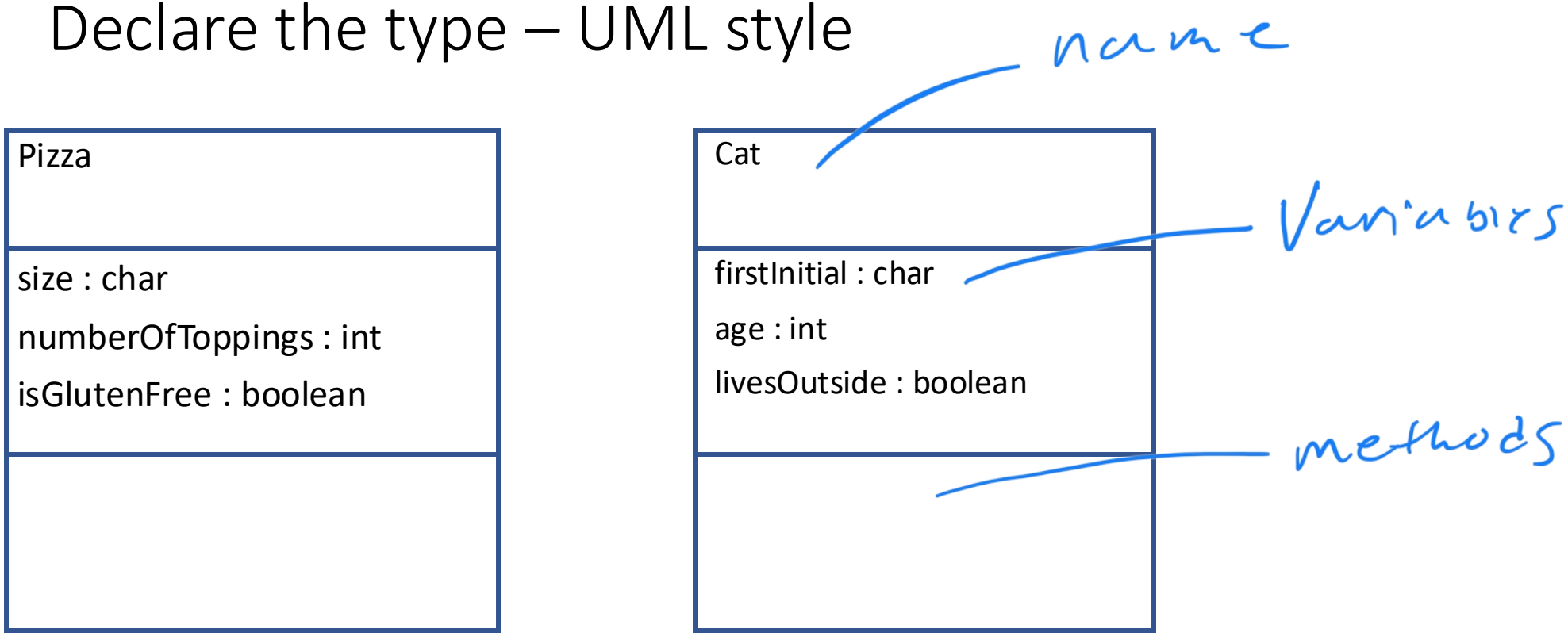
Telling Java what data type the variable is

Java is strongly typed

- Conserve space
- Can’t change – “built to order storage”
- Declaring a variable means telling java what type that variable is

- First time Java sees a variable, it must be declared!!!

Declare the type – UML style



Visibility Modifiers

What do you want to be “hidden” from others?

Visibility Modifiers

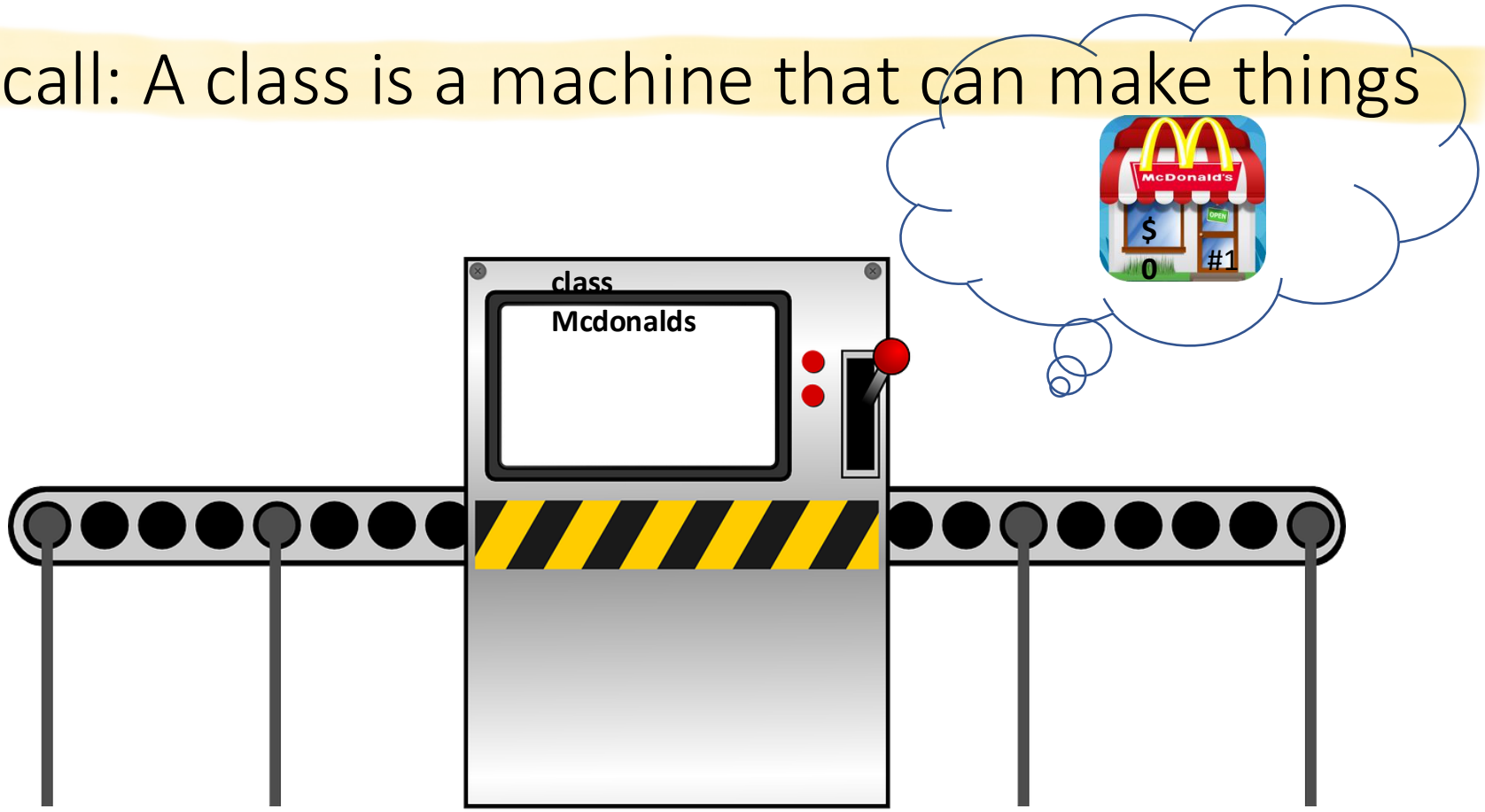
Package

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes *(Package, Inheritance)*	Yes *(Package)*	No
From a subclass outside the same package	Yes	Yes *(Inheritance)*	No	No
From any non-subclass class outside the package	Yes	No	No	No

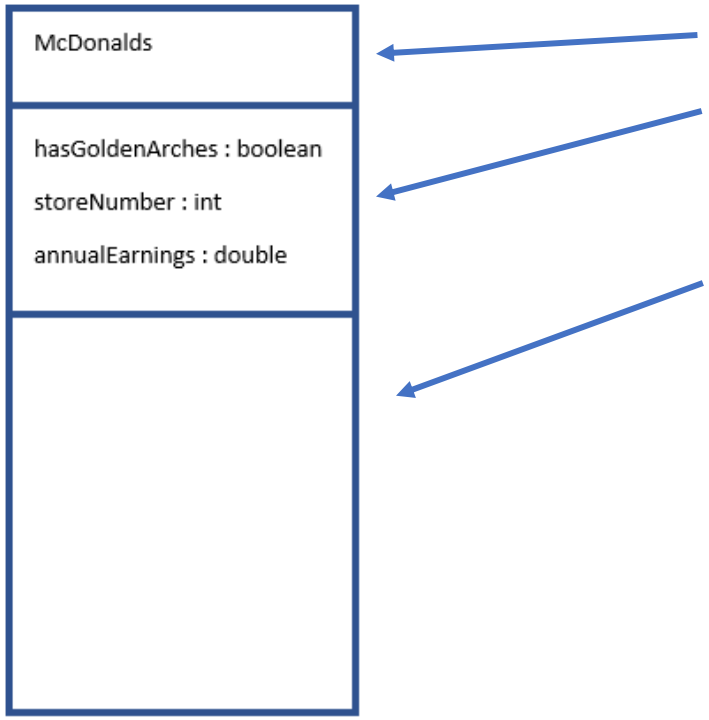
Classes

Review what we know so far

Recall: A class is a machine that can make things



Recall UML Diagram



Noun (object)

Adjectives (instance variables)

Describe the state of the object

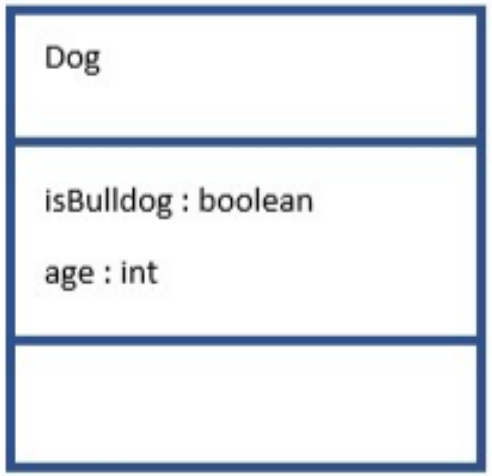
Verbs (methods)

Describe what the object can do

6-Try it!

- Open Eclipse and create a project called DogApplication

- Create a class based on the following UML:



Driver class

Has main method.

Boss class

- Runs everything. Pushes buttons on machines to make objects
- Usually boss doesn't do any "work" – just tells everyone else to
- Don't use word boss – instead use word driver

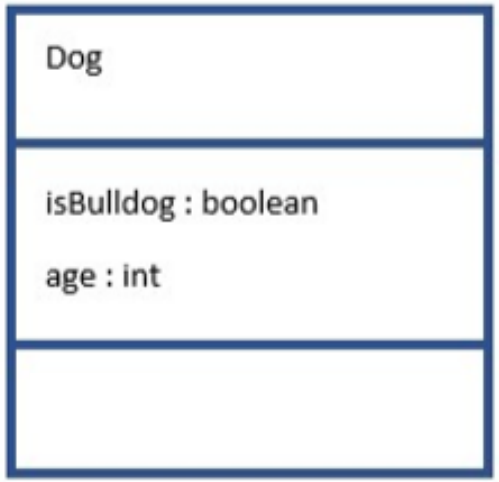
- Could have main method inside small classes
  - One person organization – serve multiple roles

McDonalds.java | *McDonaldsDriver.java

```
1
2  public class McDonaldsDriver {
3
4      public static void main(String[] args) {
5
6
7      }
8
9  }
10
```

7-Try it!

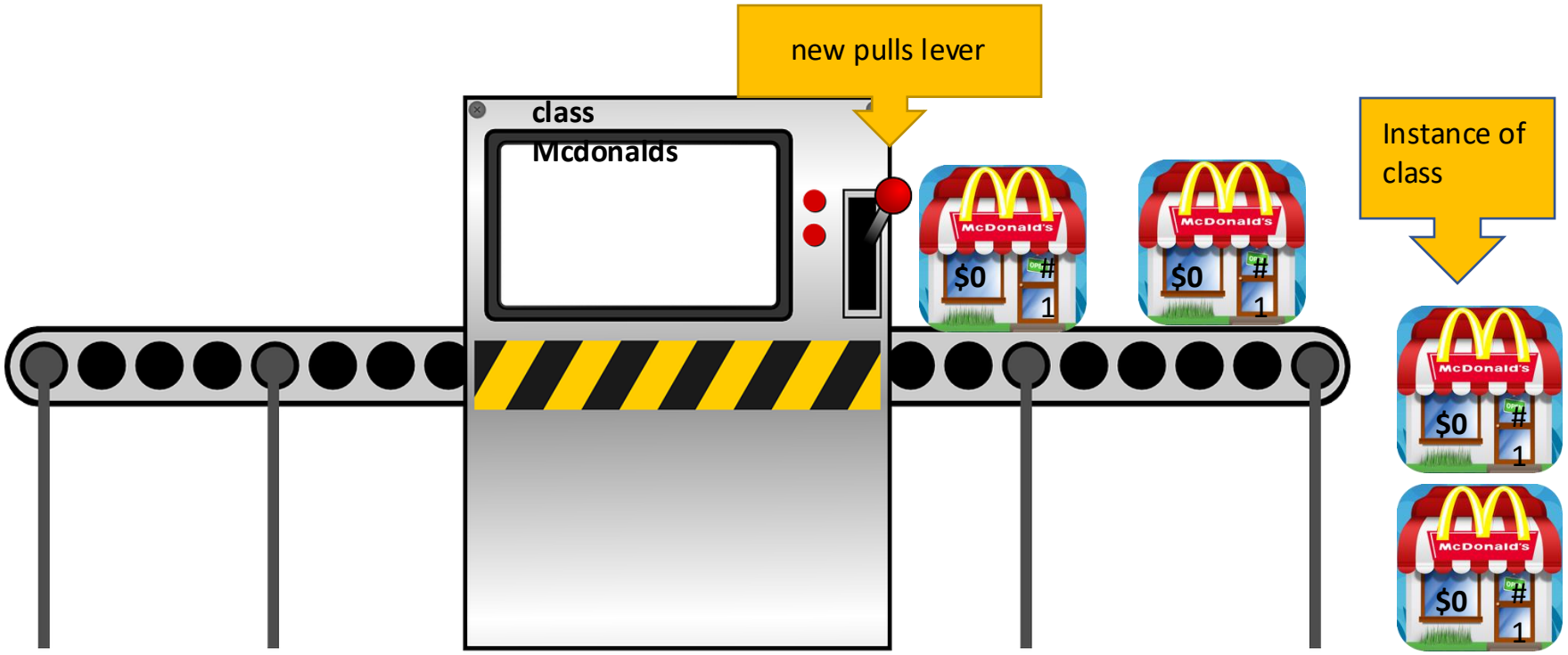
- Add a driver class to the DogApplication project



Instantiating an Object

Using “new”

Use new to instantiate



McDonalds.java | *McDonaldsDriver.java

```
1
2  public class McDonaldsDriver {
3
4      public static void main(String[] args) {
5          //Notice no privacy modifiers because we are inside a method.
6          McDonalds store1 = new McDonalds();
7          McDonalds store2 = new McDonalds();
8          McDonalds store3 = new McDonalds();
9          McDonalds store4 = new McDonalds();
10
11     }
12
13 }
14
15
```

Constructors

A constructor helps you “build” the object

Constructor (Assign values to variables)

- Usually give initial values to variables (initialize) with a special method called a constructor (can tell it is a constructor because it has the same name as the class and no return type)
  - Not ideal to use methods (including setters and getters) in constructor
  - Assignment occurs with =
- The constructor is called during instantiation
- No-argument (no-arg) constructor has no parameters
- Default constructor is a no-arg constructor that sets everything to “nothing”
  - Once you code a constructor, there is no longer a default constructor

*McDonalds.java *McDonaldsDriver.java

```
1
2  public class McDonalds {
3      private boolean hasGoldenArches;
4      private int storeNumber;
5      private double annualEarnings;
6
7      public McDonalds() {
8          hasGoldenArches = true;
9          storeNumber = 1;
10         annualEarnings = 0;
11     }
```

Constructors that accept values / Overloading methods

- Want to be able to initialize with different values
  - McDonalds in Sedona, AZ with teal arches

- Methods can have same name as long as they accept different parameters.
  - public McDonalds ()
  - public McDonalds (int storeNum)
  - public McDonalds (boolean golden, int storeNum, double earnings)
  - If have above, can't have public McDonalds (int earnings)

Constructors that accept values / Overloading methods

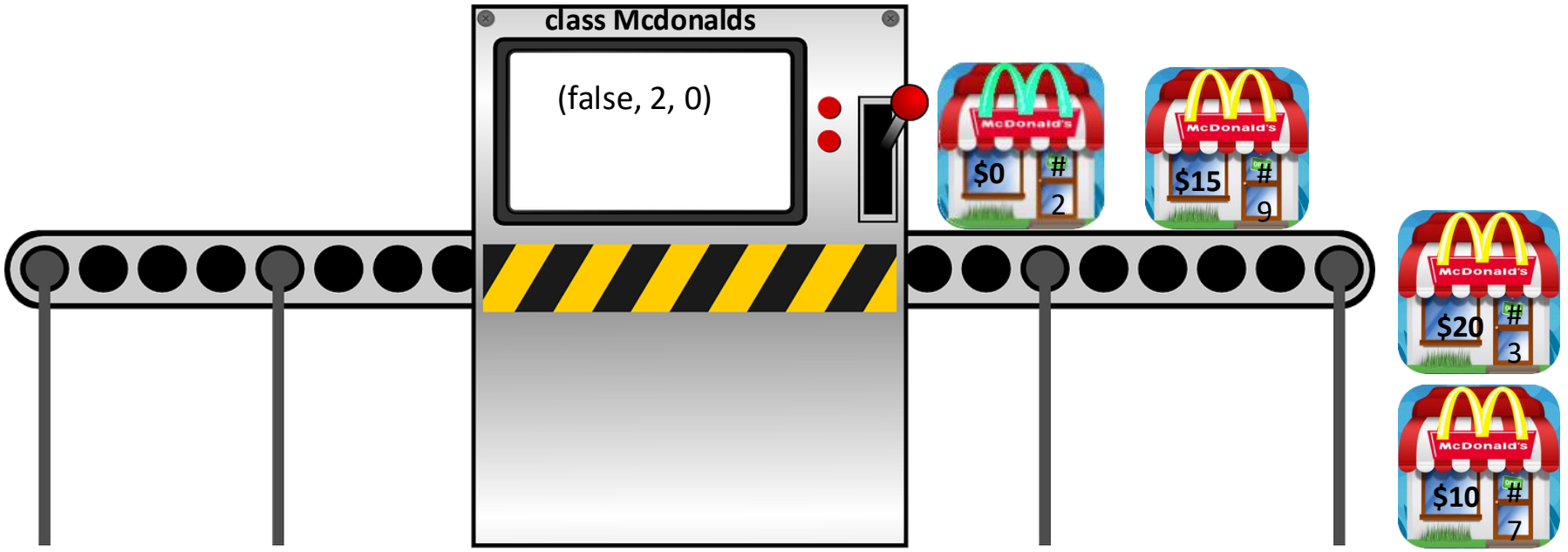
- Want to be able to initialize with different values
  - McDonalds in Sedona, AZ with teal arches

- Methods can have same name as long as they accept different parameters.
  - public McDonalds ()
  - public McDonalds (int storeNum)
  - public McDonalds (boolean golden, int storeNum, double earnings)
  - If have above, can't have public McDonalds (int earnings)

Both constructors can't have the same type and number of parameters. Here both are accepting int.

Java isn't reading the variable name when searching for a match, just the variable type

Send inputs to the constructor



*McDonalds.java *McDonaldsDriver.java

```java

public class McDonalds {
    private boolean hasGoldenArches;
    private int storeNumber;
    private double annualEarnings;

    public McDonalds() {
        hasGoldenArches = true;
        storeNumber = 1;
        annualEarnings = 0;
    }

    public McDonalds(boolean has, int num, double earnings){
        hasGoldenArches = has;
        storeNumber = num;
        annualEarnings = earnings;
    }

```

8-Try it!

1. Create a constructor in the Dog class that takes two input parameters and uses those parameters to initialize the instance variables.
2. In the main method inside the driver class, declare a variable of type Dog (call it myDog) and instantiate it with values of your choice.
3. In the main method inside the driver class, declare a variable of type Dog (call it yourDog). Instantiate it with appropriate values so that it represents a 2 year old bulldog.

METRO STATE UNIVERSITY

Abstract Data Types – Classes (2)

Primitive Data Types

Abstract Data Types (ADT)

Let’s try to define as Abstract Data Type…

- Find a way to express the states or store data about the new data type
- Define the set of operations that can be done on the new data type
  - We use methods
    - Initialization -- Constructors
    - Obtain the values of the various fields -- Getters
    - Modify the fields -- Setters
    - Compare different instances of the same object

Let’s try to define as Abstract Data Type…

- Find a way to express the states or store data about the new data type
- Define the set of operations that can be done on the new data type
  - We use methods
    - Initialization -- Constructors
    - Obtain the values of the various fields -- Getters
    - Modify the fields -- Setters
    - Compare different instances of the same object

Accessors and Mutators

Official term for getters and setters

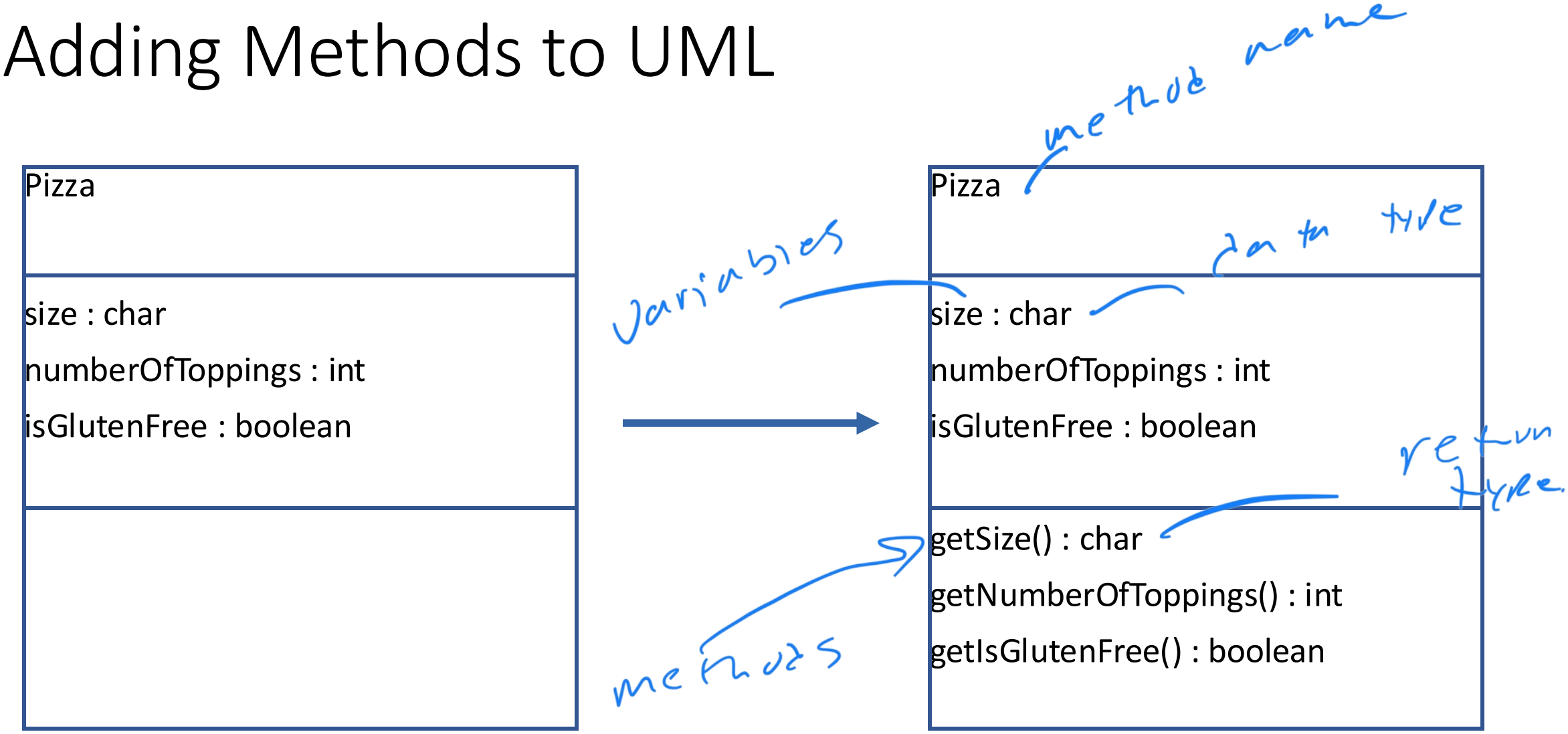
Great way to practice creating methods – everything is predetermined

Getters (Accessors)

- Returns the value that is stored in the variable
  - recall that we have hidden the information by using the private modifier
  - the getter “gets” or “allows access to” the information for others

- Always written as getVariableName()

Adding Methods to UML

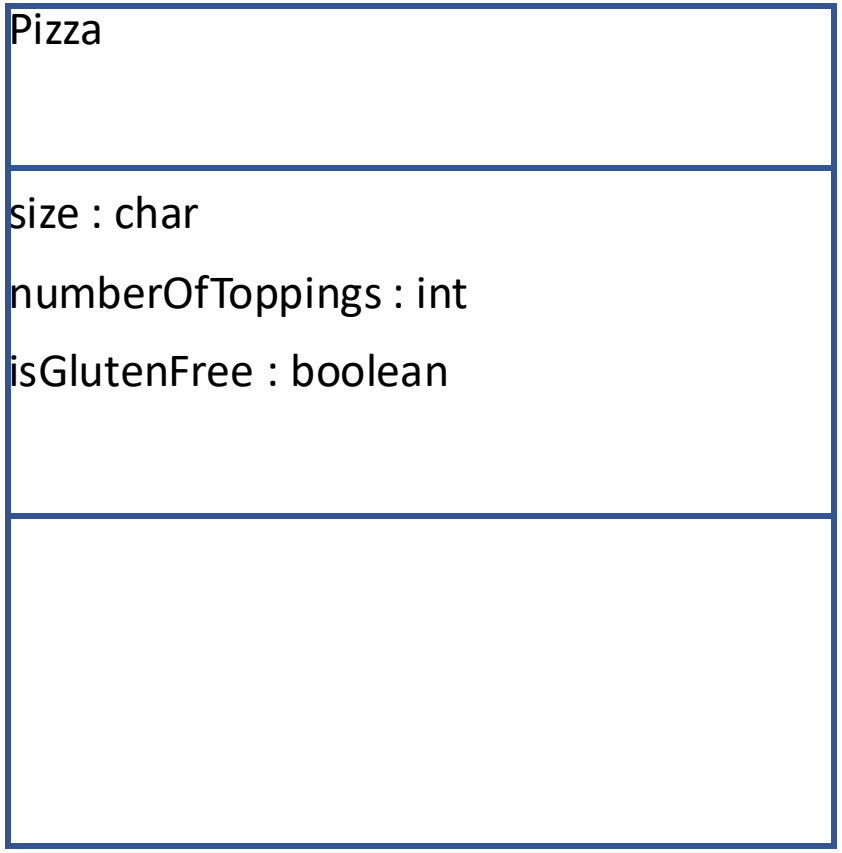


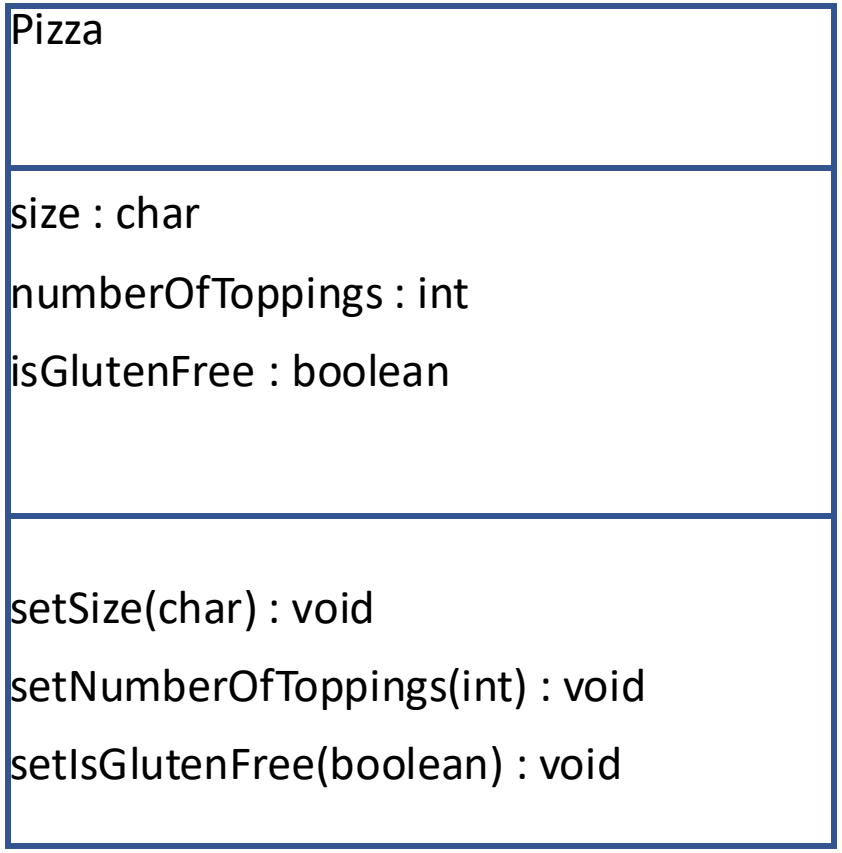
Setters

- Changes the value stored in the variable
  - the setter allows others to “set” or “mutate” the value stored in the variable

- Always written as setVariableName(parameter)

Add method to UML





Activity:

- Add a getter for all of the variables of the dog class.
- Add a setter for all of the variables of the dog class.

Class – What we've learned so far...

Review parts of a “basic” Class

- **variables** to hold the current state of the object
- **constructor(s)** to help build the object
- **getters/setters** as needed to access information about the object’s current state

Review parts of a “basic” Class

- **variables** to hold the current state of the object
- **constructor(s)** to help build the object
- **getters/setters** as needed to access information about the object’s current state
- **toString()** to describe the object in a way humans understand
- **equals()** to decide if two objects of the same type are equal

toString()

toString()

- What happens when you print a primitive data type?
- What happens when you try to print an object?

- Remember that the name of object is a reference

- What do you want it to print when you are printing the object?
  - If you had to write one sentence to describe your thing, what would it say?
  - Design decision

Example of *toString()* method for the McDonalds class

*McDonalds.java *McDonaldsDriver.java

```
1
2  public class McDonalds {
3      private boolean hasGoldenArches;
4      private int storeNumber;
5      private double annualEarnings;
6
7      public McDonalds() {
8          hasGoldenArches = true;
9          storeNumber = 1;
10         annualEarnings = 0;
11     }
12
13     public McDonalds(boolean has, int num, double earnings){
14         hasGoldenArches = has;
15         storeNumber = num;
16         annualEarnings = earnings;
17     }
18
```

```
    public String toString(){
        String output = "";
        output += "The manager of store number " + storeNumber;
        output += " is " + managerName;
        return output;
    }
```

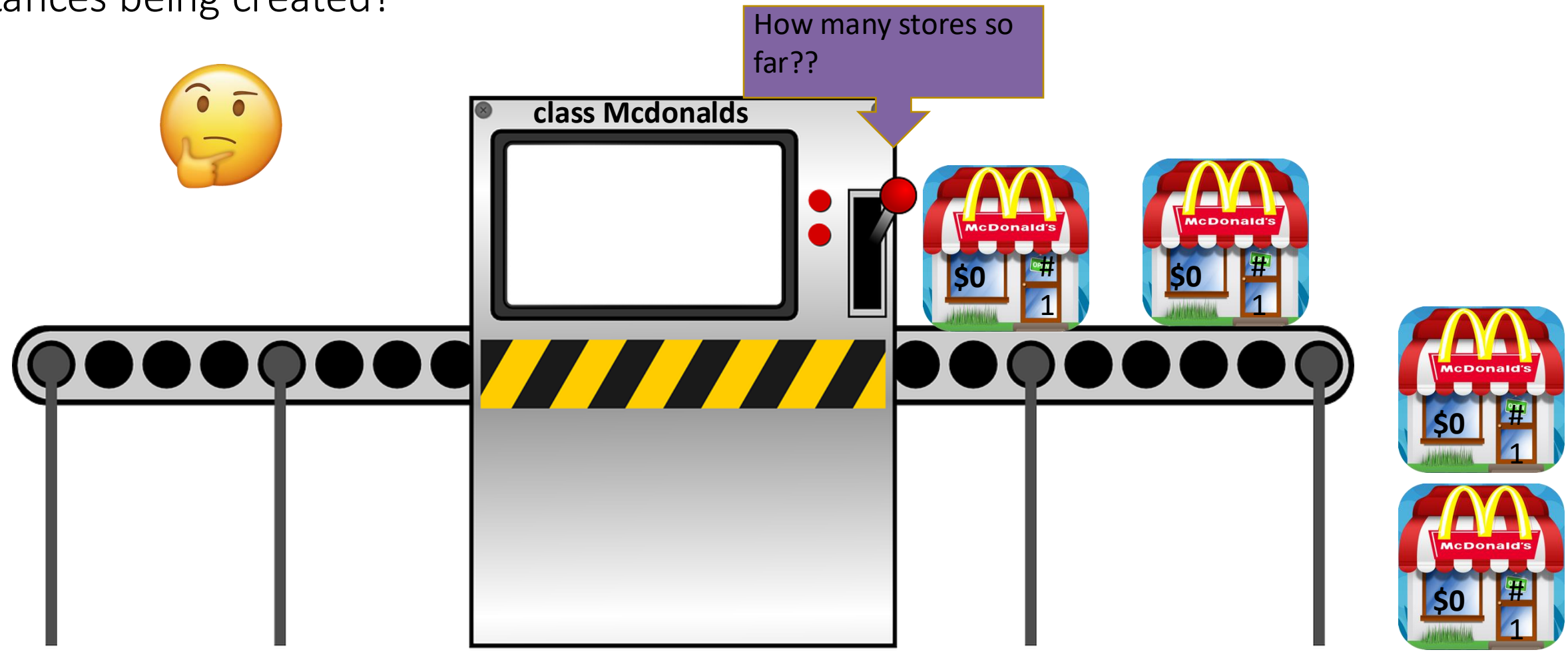
Activity

- Create a *toString()* method for your Dog class. What sentence will you use to describe the dog?

Review of Instance Fields and Methods

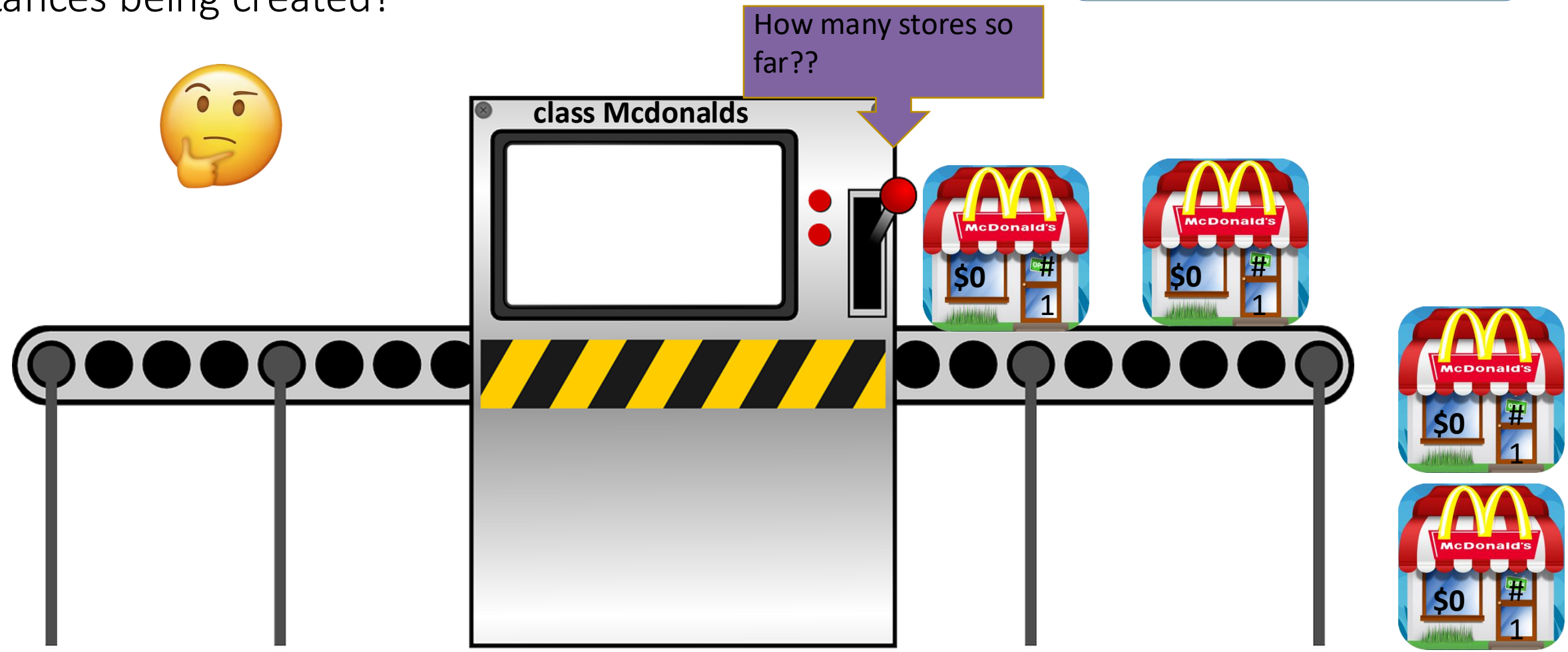
- Each instance of a class has its own copy of instance variables
- Instance methods require that an instance of a class be created in order to be used.
- Instance methods typically interact with instance fields or calculate values based on those fields.

What if we want to keep track of the number of instances being created?



What if we want to keep track of the number of instances being created?

We can use a field with the **static** modifier



static modifier

Static Class Members

- Any member with the ***static*** modifier belongs to the class itself
  - ***Static fields*** and ***static methods*** do not belong to a single instance of a class.

- To invoke a static method or use a static field, the class name, rather than the instance name, is used.

Static Fields

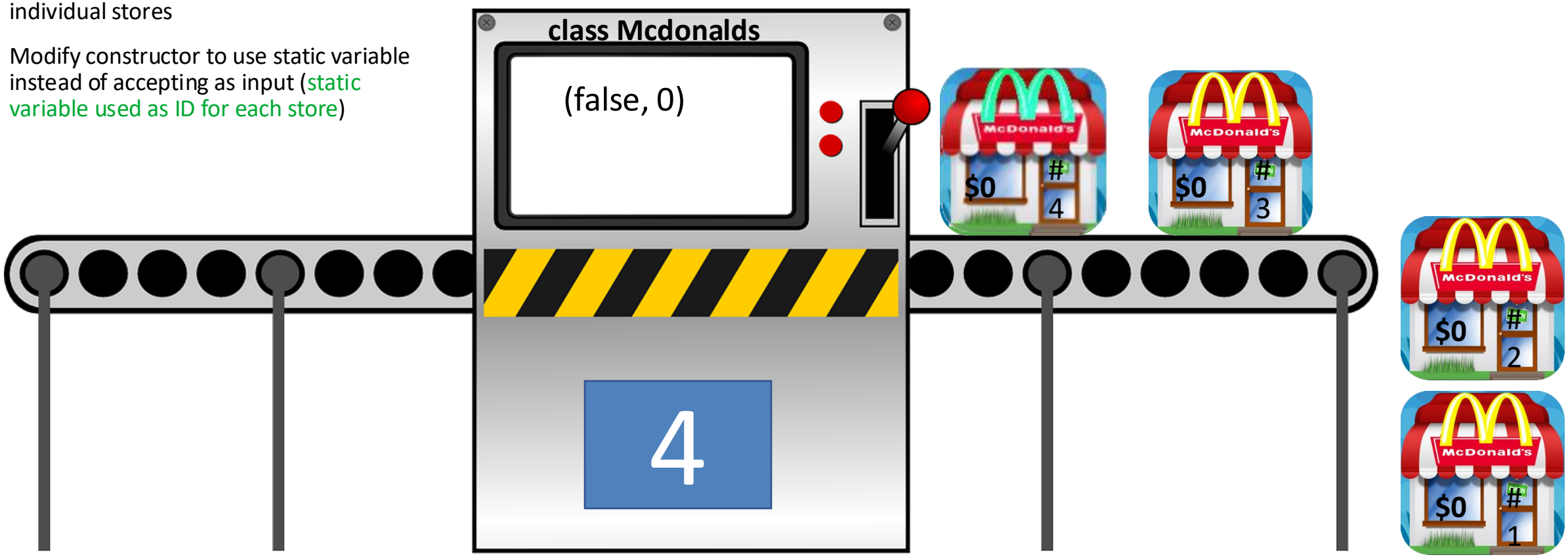
- Class fields are declared using the static keyword between the access specifier and the field type:

```
private static int staticVariableName = 0;
```

- The field is initialized to 0 **only once**, regardless of the number of times the class is instantiated.
  - Primitive static fields are initialized to 0 if no initialization is performed.

Static

- Want to keep track of number of stores (for store number)
- Belongs to the machine, not the individual stores
- Modify constructor to use static variable instead of accepting as input (static variable used as ID for each store)



```java
public class McDonalds {
    private boolean hasGoldenArches;
    private int storeNumber;
    private double annualEarnings;
    private String managerName;
    private static int numStores = 0;

    public McDonalds(double earnings) {
        hasGoldenArches = true;
        numStores = numStores + 1;
        storeNumber = numStores;
        annualEarnings = earnings;
        managerName = new String("Jess");
    }

    public McDonalds(boolean golden, double earnings, String name) {
        hasGoldenArches = golden;
        storeNumber = ++numStores;
        annualEarnings = earnings;
        managerName = name;
    }
```

Here we use the static variable to initialize the store number in the constructor

```java
public class McDonalds {
    private boolean hasGoldenArches;
    private int storeNumber;
    private double annualEarnings;
    private String managerName;
    private static int numStores = 0;

    public McDonalds(double earnings) {
        hasGoldenArches = true;
        numStores = numStores + 1;
        storeNumber = numStores;
        annualEarnings = earnings;
        managerName = new String("Jess");
    }

    public McDonalds(boolean golden, double earnings, String name) {
        hasGoldenArches = golden;
        storeNumber = ++numStores;
        annualEarnings = earnings;
        managerName = name;
    }
```

Static Methods

- Methods can also be declared static by placing the `static` keyword between the access modifier and the return type of the method.

```
public static returnType staticMethodName(paramList)
{…}
```

- When a class contains a static method, it is not necessary to create an instance of the class in order to use the method.

```
returnType varName = ClassName.staticMethodName(argList);
```

Static Methods

- Static methods are convenient because they may be called at the class level.
  - They are typically used to create utility classes, such as the `Math` class in the Java Standard Library.

- Static methods may not communicate with instance fields, only static fields.

- Static methods belong to the machine, so have to ask the machine (class)

- Methods belong to individual objects, so have to ask an individual object

Activity:

- Add a static counter to the dog class.
- Create an variable to hold the dog’s id.
- Update the constructor(s) to use the static variable to assign a unique id to every dog that is created.

Comparing Objects

The `equals()` Method

Primitive types

- Primitive types are so simple they cannot be broken down into smaller pieces
- Java provides operators for comparing variables which contain primitive types

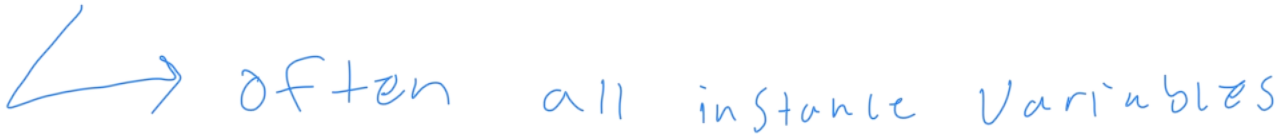
  `<` `>` `==` `<=` `>=` `!=`

- These operators **only work properly on primitive types**

```
int age = 13;
if (age > 20)
   System.out.println("Adult");
```

Comparing Two Objects

- What does it mean to compare two objects?
- How do you know if they are equal?
- Or if one object is less than another object?
- Remember: Objects are COMPOSITE types
- You must write a method to compare two objects
- Choose the most important instance variable(s) and use it for comparison



How do you know if two things are equal?

- Depends on what type of object you are talking about:
  - Primitive data types
    - 2, ‘a’,
    - double (iffy) Correct to the Unit of least percision
    - ==
  - Objects:
    - McDonalds
    - Dog
    - String
    - .equals(Object other) …… actually using the this reference
- Programmer gets to decide what makes two of their objects equal
  - One of the objects is always the thing that called the method, the other is sent as a parameter.
  - When the call is made, essentially asking “am I equal to this other thing”
    - Returns true if yes
    - Returns false if no

Example: When Are Two Points Equal?

- Two points are considered equal if their x and y coordinates match

```
if (this.x == otherPoint.x
   && this.y == otherPoint.y)
         return true;
```

`equals()`

- The `equals` method has a standardized format. The header always looks like this

```
public boolean equals (Object o)
```

- `equals` tests to see if the Object is of the expected type

```
if (o instanceof MyPoint)
```

- If two objects are of different types, they cannot be equal

equals for class Point

```
public class Point{
          private int x;
          private int y;

          public Point(int x, int y){
                    this.x = x;
                    this.y = y;
          }
          @Override
          public boolean equals(Object other){
                    boolean output = false;
                    if (other instanceof Point){
                              Point otherPoint = (Point) other;
                              if (this.x == otherPoint.x && this.y == otherPoint.y)
                                        output = true;
                    }
                    else
                              output = false;
                    return output;
          }
}
```

Calling `equals`

```
Point p1 = new Point(20,30);
Point p2 = new Point(10, 20);
Point p3 = new Point(10, 20);

System.out.println(p1.equals(p2));  // false
System.out.println(p3.equals(p2));  // true
```

Calling `equals`

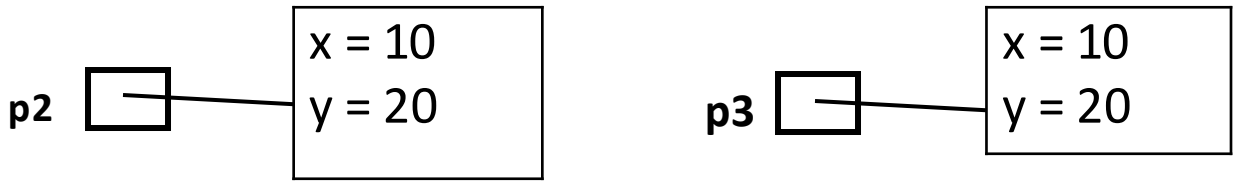
```
System.out.println(p1.equals(p2));
```

- The object to the left of the dot operator is called the invoking object
- The object inside the parentheses is called the parameter (or argument)
- Inside the `equals` method
  - The invoking object can be referred to as ***this***
  - The formal parameter name (o) is used to refer to the actual parameter (p2)

==

```
Point p2 = new Point(10, 20);
Point p3 = new Point(10, 20);
System.out.println(p3==p2);
```

- == compares the contents of the **reference variables** and not the contents of the objects



- p2 != p3 because they do not point to the same object

Moral

- Always use a method to compare two objects
- The programmer decides what it means for two objects to be considered equal

Another Example:McDonalds had a unique id – storeNumber

```
public class McDonalds {
    private boolean hasGoldenArches;
    private int storeNumber;
    private double annualEarnings;
    private static int numStores = 0;


    public boolean equals(McDonalds other){
        return this.storeNumber == other.storeNumber;
    }
```

Activity

- Add an equals() method to the dog class. Two dogs should be considered equal if they have the same id.

Using the Classes -

Seeing objects

“Seeing” objects

- Almost everything in Java is an object.

field ≡ instance variable ≡ attribute

- These objects have fields and methods.

- We use the objects, fields, and methods to accomplish our goals.

- Sooner see objects, easier it is to code in Java. Not memorizing anything, using an objects abilities.

Dot Notation

Dot notation

instance.field or instance.method()

ClassName.staticFieldName or ClassName.staticMethodName() (for static field or static method)

For example, if we have a class Student, and we made the following student:

Student aStudent = new Student();

Then we could write:

aStudent.goToClass()

aStudent.gpa

or

Student.numStudents (only possible if numStudents is NOT a private field)

If variable is private, can't use instance.field X Can't use outside of class

- That's why we have getters and setters

McDonalds.java | *McDonaldsDriver.java

```
1
2  public class McDonaldsDriver {
3
4      public static void main(String[] args) {
5          //Notice no privacy modifiers because we are inside a method.
6          McDonalds store1 = new McDonalds(true, 0);
7          McDonalds store2 = new McDonalds(true, 0);
8          McDonalds store3 = new McDonalds(true, 0);
9          McDonalds store4 = new McDonalds(true, 0);
10
11         double earnings = store1.annualEarnings;  //wrong for private variable - fine for public
12
13         double earningS = store1.getAnnualEarnings();  //right
14
15         store1.annualEarnings = 1000;  //wrong for private variable - fine for public
16
17         store1.setAnnualEarnings(1000); //right
18     }
19
20 }
21
```

Wrapper Classes

Wrapper Classes

- Many methods are designed to work with objects.

- Ideally, you could write one method that would work with any type of object.
  - But then how can you get it to work with primitives?

- Solution: You "wrap" the primitive in an object.
  - The class from which that object would be created is a wrapper class

Wrapper Classes

- A wrapper class stores an entity (the *wrapee*) and adds operations that the original type does not support correctly

- Each wrapper object is:
  - *immutable* (meaning its state can never change),
  - stores one primitive value that is set when the object is constructed, and
  - provides a method to retrieve the value.

- Java provides one wrapper class per primitive:

Primitive	byte	short	int	long	float	double	char	boolean
Wrapper	Byte	Short	Integer	Long	Float	Double	Character	Boolean

Wrapper Classes

- A wrapper class stores an entity (the *wrapee*) and adds operations that the original type does not support correctly

- Each wrapper object is:
  - *immutable* (meaning its state can never change),
  - stores one primitive value that is set when the object is constructed, and
  - provides a method to retrieve the value.

- Example: Integer class

https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html

♦ **Constructor for the Integer class**

`public Integer(int value)`

Construct an `Integer` object from an `int` value.

Parameter:

`value` – the `int` value for this `Integer` to hold

Postcondition:

This `Integer` has been initialized with the specified `int` value.

♦ **intValue**

`public int intValue( )`

Accessor method to retrieve the `int` value of this object.

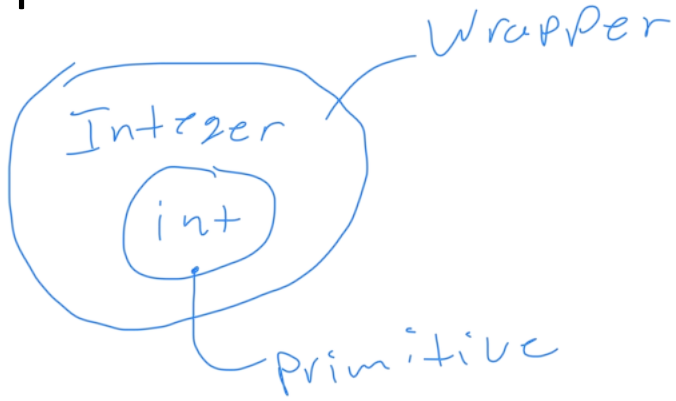
Returns:

the `int` value of this object

Boxing and Unboxing Primitives

- Boxing = converting from primitive type to object
- Unboxing = conversion from object type to primitive

```
int j = 17;
Integer k = Integer.valueOf( j );   // Boxing
int m = k.intValue();               // Unboxing
```



- Boxing and unboxing can be automatic in some cases.
  - If Java expects an object of a wrapper class and it gets a primitive of the appropriate type, it autoboxes.
  - If Java expects a primitive and it gets a wrapped primitive of the appropriate type, it auto-unboxes.

```
int j = 17;
Integer k = j;     // Autoboxing
int m = k;         // Autounboxing
```

Advantages & Disadvantages of Wrapper Classes

- Main advantage: allows to treat primitives like objects
- Main Disadvantage: ordinary primitive operations are no longer directly available
- Example: suppose x , y , and z are Integer objects. The statement z = x + y is still valid but its evaluation is now slow because:
  - First, x and y must be auto-unboxed for '+' to work
  - Then the result of x+y mut be autoboxed before it can be stored in z

Activity:

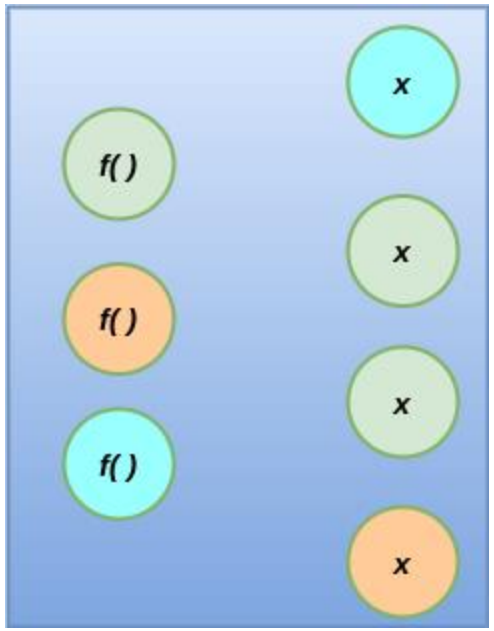
- Suppose that x , y and z are all Double objects. Describe all the boxing and unboxing that occurs in the assignment z = x + y

Object Oriented Programming (OOP) Concepts

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

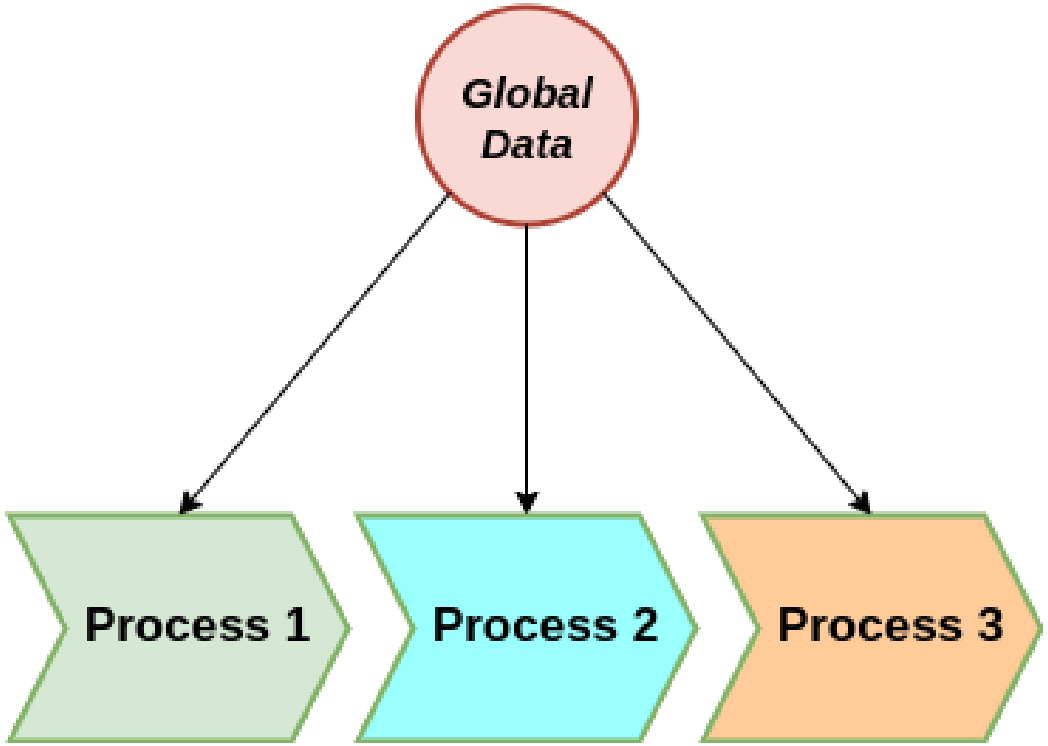
Procedural Programming

- Data is separate from methods
  - Data is usually global
  - Modified from one function to the next
- Functions are also separate
  - Can be complex
  - Usually requiring multiple parameters since they are distinct entities from data



Procedural Programming (2)

- How it usually works:

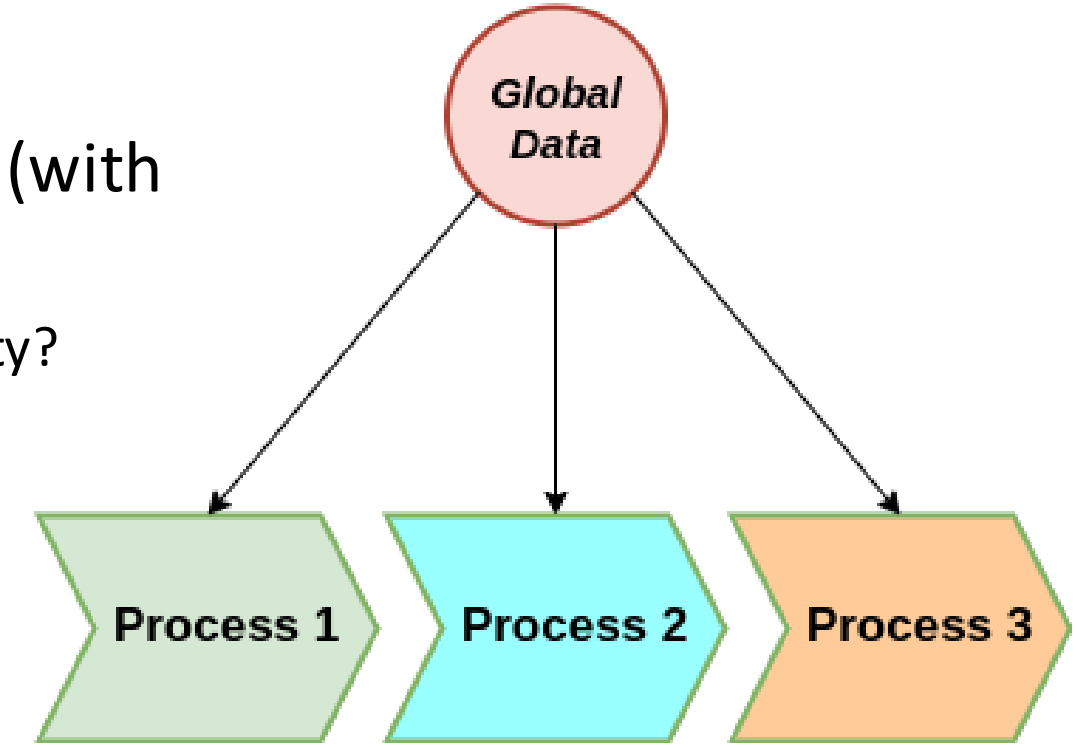


Procedural Programming (3)

- Advantages:
  - Easy to write (especially if short program)
  - Beginner-friendly

Procedural Programming (4)

- Disadvantages:
  - Not always appropriate for large program
  - Hard to represent a single “thing” (with behavior and states)
    - Can we represent a “person” as an entity?
    - Bank Account?
    - Car?

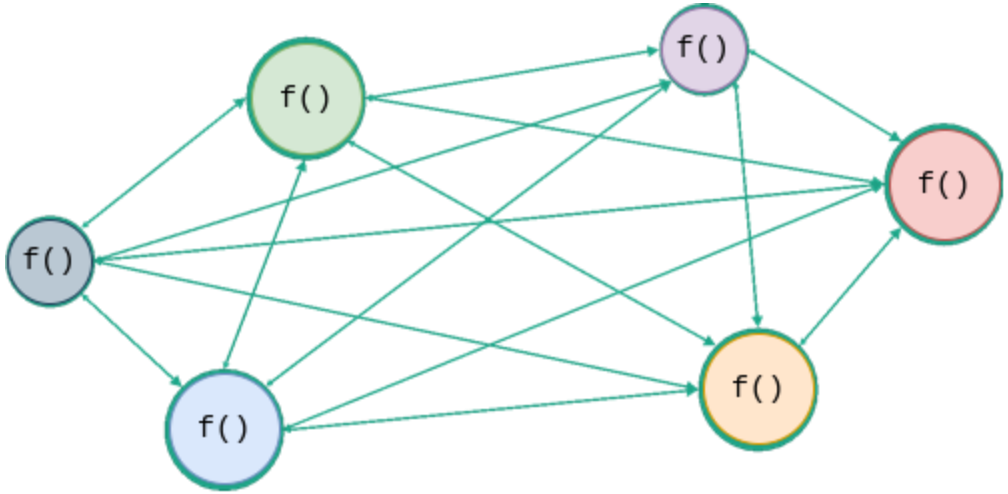


Procedural Programming (5)

- Disadvantages:
  - Immutability problem
    - Ex: would it be good to modify the balance of a bank account from anywhere in the program?
    - What about your SSN?

Procedural Programming (6)

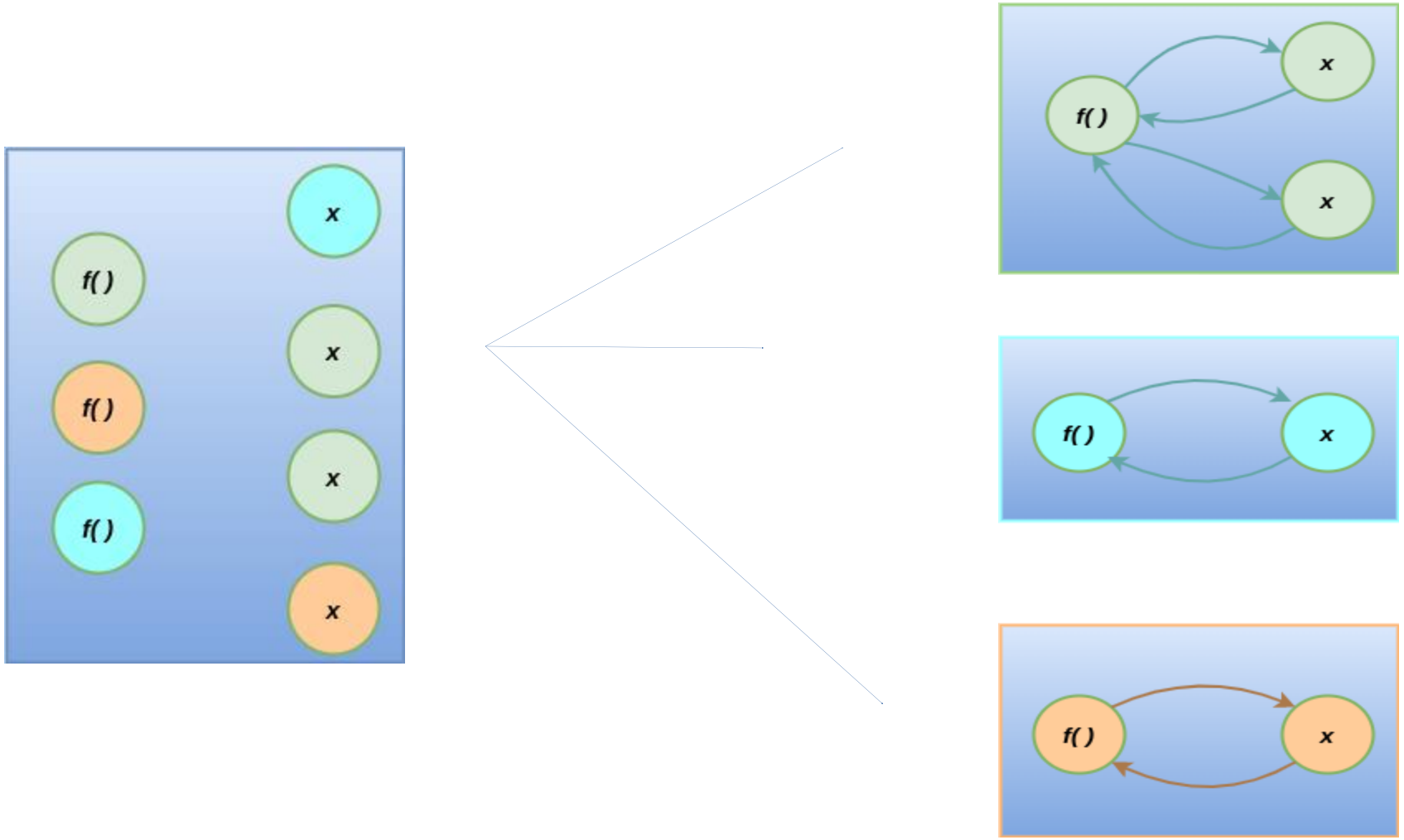
- Disadvantages:
  - Hard to maintain the entire program
  - Hard to assign portions of the code to different teams
  - One change can disrupt other modules



Object-Oriented Programming

- Centered around manipulating objects
  - Created from classes
  - Allows to bind data and methods
- Advantages:
  - Can represent abstract things

Object-Oriented Programming



Object-Oriented Programming Pillars

- Increase reusability
- Reduce complexity

- Reduce Complexity
- Isolate the impact of change (pre/post conditions, information hiding)

- Eliminate code redundancy (child class can inherit parent class methods, behaviors, states) example: employee, students are also a person

- Code can take on multiple forms