# Classes & Methods

# *Overview*

**Record Types** allow us to create and manipulate real or imaginary world entities

- Records are immutable

- Records are defined by the *properties they store,* not by the *behaviors they exhibit.*

In this module, we will learn how to create and manipulate real or imaginary world entities as **objects**, which have both *properties* AND *behaviors*.

# *Learning Objectives*

- To be able to write and use a class

- To be able to write a class constructor

- To be able to write comments

- To be able to understand and write accessor and mutator methods

- To be able to write methods

- To be able to use static variables and methods

- To be able to understand variable scope

- To be able to understand and use the `this` keyword

# *Introduction*

A **CLASS** is a template for creating objects

- (like a **RECORD TYPE** defines what **RECORDS** of that type look like)

A class defines a new **data type**

- (like record types do.)

A class defines the object's *attributes / properties* and *behaviors*

- Object's properties are implemented as **INSTANCE VARIABLES**
- Object's behavior are implemented as **METHODS**

Objects are instances of a class, the way that records belong to a record type.

3

# *Class Design*

**Abstraction**: set of information properties relevant to a stakeholder about an entity

**Information Property** (or just "property"): a named, objective and quantifiable aspect of an entity

**Stakeholder**: a real or imagined person (or a class of people) who is seen as the audience for, or user of the abstraction being defined

# *Class Design Exercise*

Entity: Movie from the point of view of someone shopping for a movie on an online storefront

**Overall Pick** ⓘ

### The Worst Person in the World (The Criterion Collection) [Blu-ray]

⭐⭐⭐⭐⭐ ⌄ 183

**Blu-ray**

$**27**$^10 List: ~~$39.95~~

✓**prime** Two-Day
FREE delivery **Wed**

More Buying Choices
$20.78 (16 used & new offers)

**DVD**

$**16**$^81 List: ~~$29.95~~

✓**prime** One-Day
FREE delivery **Tomorrow**
Only 15 left in stock (more on the way).
More Buying Choices
$9.95 (20 used & new offers)

- Starring: Renate Reinsve , Anders Danielsen Lie , Herbert Nordrum , et al.
- Directed by: Joachim Trier

# *Class Design*

Entity: Movie

Properties:

- Title

- Year

- Length

- Genre

- Format

- Price

# *Class Design*

Entity: Movie

Properties:

- Title (`String`)
- Year (`int`)
- Length (`int`)
- Genre (`String`)
- Format (`String`)
- Price (`double`)

*Instances of the Movie Class*

| Movie | | | On-Line Customer | | |
|---|---|---|---|---|---|
| Title (string) | Year (int) | Length (int) | Genre (string) | Format (string) | Price (double) |
| "Moneyball" | 2011 | 133 | "Sports" | "Blueray" | 15.00 |
| "Gone With the Wind" | 1939 | 219 | "Drama" | "DVD" | 10.95 |
| "Jurassic Park" | 1993 | 127 | "SciFi" | "DVD" | 12.50 |
| "Pirates of the Caribbean" | 2003 | 143 | "Comedy" | "Blueray" | 17.50 |
| "Sicko" | 2007 | 116 | "Documentary" | "Streaming" | 11.75 |

**Representing the Movie Abstraction using a Table**

# *Content of a Class*

A class contains

- Instance variables representing the properties of the abstraction

- One or more **constructor(s)** to initialize the objects' instance variables

- Methods to implement the objects' behaviors

# Anatomy of a Class

```java
public class Person {
  // instance variables
  private String name;
  private String email;
  private String phoneNumber;

  // constructor
  public Person(String newName, String newEmail, String newNumber) {
    // implementation withheld for now
  }

  // methods!
  public void print() {
    // implementation withheld
  }
  public void updateEmail(String newEmail) {
    // implementation withheld
  }
}
```

# *Instance Variables*

- Listed at the top of the class definition

- To declare an instance variable, you write

  ```
  private DataType variableName;
  ```

  e.g. `private String name;` or `private double price;`

- `private` means that only this class has access to this instance variable

  - important for class design! let each class only know about what it needs to.

  - opposite of `public`, which is more appropriate for functions/methods.

# *Instance Variables*

Recalling the properties we decided on for Movies...

```java
public class Movie {

    private String title;
    private int year;
    private int length;
    private String genre
    private String format
    private double price
    ...
}
```

# *Instance Variables*

Another example, for a `Person` class:

```java
public class Person {

    private String name;
    private String email;
    private String phoneNumber;
    ...
}
```

From the point of view of a University

Directory entry.

# *Instance Variables*

- Instance variables are the properties of the object

- They are in scope throughout the entire class!
  - Can be used in other functions of the class

- Usually they are not declared with an initial value
  - The initial value is usually assigned in the constructor
  - Different objects of this class will have different values for these variables!

# *Constructors*

- Set the initial values for the object's instance variables

- Constructors must have the same name as the class

- Constructors have no return type!

- To define a constructor, you write:

```
public ClassName(DataType1 parameter1, DataType2 parameter2, …){

    /* instance variable initialization */

}
```

(the parameter list can also just be empty!)

# *No-argument constructor*

Default constructor (provided by Java) initializes instance variables to default values

- Not often very useful

- `String` and array instance variables set to `null`—dangerous!

- Primitives set to `0`/`false`

equivalent to the following:

```java
public Person() {
    name = null;
    email = null;
    phoneNumber = null;
} // ¯\_(ツ)_/¯ (not very useful!)
```

# *Argument constructor*

A constructor can take in arguments to initialize the instance variables.

```java
public Person(String initName, String initEmail, String initPhone) {
    name = initName;
    email = initEmail;
    phoneNumber = initPhone;
}
```

This constructor says: "*I will create a new* `Person` *for you with these initial values for* `name`, `email`, *and* `phoneNumber`."

# *Argument constructor*

A constructor doesn't have to take one input per instance variable

```java
public Person(String initName, String initEmail) {
   name = initName;
   email = initEmail;
   if (email.indexOf("upenn.edu") != -1) {
     phoneNumber = "215-898-5000";
   } else {
     phoneNumber = "";
   }
}
```

What does this constructor do?

# *Argument constructor*

A constructor doesn't have to take one input per instance variable

```java
public Person(String initName, String initEmail) {
  name = initName;
  email = initEmail;
  if (email.indexOf("upenn.edu") != -1) {
    phoneNumber = "215-898-5000";
  } else {
    phoneNumber = "";
  }
}
```

This constructor says: "*I will create a new* `Person` *for you with these initial values for* `name`, `email`. *If they have a Penn email, I'll give them a default Penn phone number. Otherwise, I'll leave the field blank with an empty String.*"

# *Methods*

**Methods** are functions that belong to objects of a class. They define how an object behaves based on its properties.

- Every object from a class has the same methods and the same instance variables.

- The values of the instance variables differ between the objects.

- $\Rightarrow$ Since methods behave differently based on the values of the instance variables, they can behave differently for different objects.

# *Special methods*

- **Accessor methods**: to retrieve and return the value of the instance variables

  - Records gave you these for free!

- **Mutator methods**: to change (update) the value of the instance variables

  - These are not possible with Records

- **Main method**: used to test your class (execute your code). There can be only one main method inside a class

# *Accessor Methods*

- AKA "getter" methods

- Used to return the value of an instance variable

- Usually take no input, have the return type of the instance variable they're getting

```
// general structure
public VariableType getVariableName(){
    return variableName
}
```

e.g.

```
public String getName(){
    return name;
}
```

# *Mutator Methods*

- AKA "setter" methods

- Used to change the value of an instance variable

- Usually take an input matching type of the variable being set, have no return type

```
// general structure
public void setVariable(VariableType v){
   instanceVariable = v;
}
```

e.g. in the Person class:

```
public void setName(String newName){
   name = newName;
}
```

# *Methods, in General*

Define the objects' behavior

- Can only be called on an object that was created using the constructor

- Can return a value or not

- To call a method, you write

```
objectName.methodName(/* parameters or not*/);
```

Example:

```
Person p = new Person();
p.setName("Mariah");
```

# *Methods, in General*

A method is just a function, so it has:

- A signature

- A body

```
public returnType methodName(/* parameters */){
    // method's body
}
```

For example:

```
public String toString() { // signature
  return "my name is: " + name; // body, uses instance variable
}
```

# *Some Common Methods*

- `public String toString()`
  - A method that lets you print out a human-readable String representation of the object
  - Java doesn't do this for you, sadly!

- `public boolean equals(Object other)`
  - A method that lets you decide if this object is the same as some other object
  - `==` is usable, but like with Strings, it doesn't do what you expect! 🚨🚨🚨

# *Methods with Parameters (Inputs)*

When calling a method with parameters, you must provide actual values for each of the **formal parameters** (inputs).

- Within `Point.java`, we might write:

```java
public void move(double dx, double dy) {
  x = x + dx;
  y = y + dy;
}
```

- This could be called by writing, for example:

```java
myPoint.move(-0.5, 1);
```

# *Methods & Primitive Inputs*

When calling a method with a *primitive type input,* any changes to the value of the variable will not be reflected outside of the method call.

```java
public void setXWithinLimit(double newX) {
    if (newX > 1) {
        newX = 1;
    }
    x = newX;
}
```

```java
public static void main(String[] args) {
    Point p = new Point(0.4, 0.3);
    double newX = 34;
    p.setXWithinLimit(newX);
    System.out.println(p.getX()); // 1
    System.out.println(newX);     // 34
}
```

# *Methods & Object Inputs*

When calling a method with an *object type input,* (any array, Point, etc.):

- An **alias**, or copy of the reference, is stored in the parameter variable

- Changes to the object inside of the method will be visible outside of the method!

# *Methods & Object Inputs*

`copyXToOtherPoint` takes a `Point` as input and modifies its `x` value.

```java
public void copyXToOtherPoint(Point other) {
  other.setX(x);
}
```

```java
public static void main(String[] args) {
    Point a = new Point(0.4, 0.3);
    Point b = new Point(0, 0);
    System.out.println(b.getX()); // 0
    a.copyXToOtherPoint(b);
    System.out.println(b.getX()); // 0.4
}
```

The modification made to `other` inside the body of `copyXToOtherPoint` is reflected in whatever object was passed in as input—`b` in this case.

# *Static Variables & Methods*

# *Static Variables & Methods*

# *Static Variables & Methods*

- Instance variables and non-static methods define the properties and behaviors of the objects of a class.
    - These variables and methods are referenced using the name of a particular object instance
    - `p.getX()`, `p.toString()` where `p` is a particular `Point` that's been initialized
- **Static** variables and methods belong to the *entire class* and do not vary among objects of the class.
    - These are referenced using the name of the class itself, e.g. `PennDraw.clear()` or `Math.random()` or `Math.PI`

# *Static Variables & Methods*

- To tag a method or variable as static, write `static` after the public/private modifier

```java
public class Point {
  private double x;
  private double y;
  private static int numPointsCreated = 0; // all Points share this value

  public Point(double newX, double newY) {
    x = newX;
    y = newY;
    numPointsCreated++; // all Points see a new value for this var.
  }
  // other methods omitted...
  public static int getNumPoints() {
    return numPointsCreated;
  }
}
```

# *Static Variables & Methods*

Later, in `main`…

```java
public static void main(String[] args) {
  Point first = new Point(0.1, 0.3);
  System.out.println(Point.getNumPoints()); // prints 1
  Point second = new Point(0, 0);
  System.out.println(Point.getNumPoints()); // prints 2

  // technically, you can do this, too:
  System.out.println(first.getNumPoints());   // prints 2
  System.out.println(second.getNumPoints());  // prints 2
}
```

# *Making Scope Explicit*

- The **scope** of a variable is where the variable is accessible by name and depends on where the variable was declared.

- Three main levels of scope when designing classes:
  - **CLASS LEVEL SCOPE**: used for instance variables, these are accessible in the entire class.
  - **METHOD LEVEL SCOPE**: used for "local" variables and method inputs, these are accessible inside of a single method.
  - **BLOCK LEVEL SCOPE**: used for loop control variables, these are accessible only inside the body of a loop or conditional

*Scoping
out Scope*