CS 5004: Lecture 1

Northeastern University, Spring 2021

Agenda

- Java vs. Python syntax basics
- Intro to object-oriented design (OOD)
- Classes in Java
- Intro to testing

Java vs. Python syntax basics

A quick overview of some key differences

- Use this as reference
- Check the docs for more

Java vs. Python – declaring variables

Python

- don’t specify type
- *can* change type
- must have a value

```
my_var … -> error
my_var = 1
my_var = “one”
```

Java vs. Python – declaring variables

Python

- don't specify type
- *can* change type
- must have a value

```
my_var … -> error
my_var = 1
my_var = "one"
```

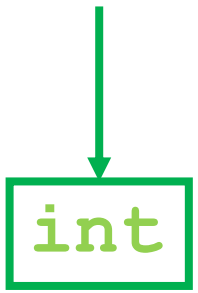
Java

- must specify type
- don't need a value right away

```
int myVar;
myVar = 1;
myVar = "one"; -> error
```

Java vs. Python – declaring variables

Type of value to be stored in the new variable

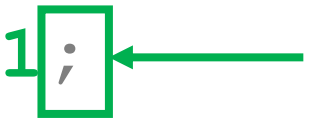


```
<datatype> <variable>;
OR <datatype> <variable> = <value>;

int myVar = 1;
```

Java vs. Python – declaring variables

```
int myVar = 1;
```



Note the semi-colon! Goes at the end of every statement.

Java vs. Python – conditionals

Python

- indentation and : required
- keywords `if, elif, else`

```
if my_var > 2:
   # do something
elif my_var < 0:
   # do something else
```

Java vs. Python – conditionals

Python

- indentation and : required
- keywords `if, elif, else`

```
if my_var > 2:
   # do something
elif my_var < 0:
   # do something else
```

Java

- indentation for readability
- `if, else if, else`

```
if (my_var > 2) {
   // do something
}
else if (my_var < 0) {
   // do something else
}
```

Java vs. Python – conditionals

```
if (my_var > 2) {
   // do something
}
else if (my_var < 0) {
   // do something else
}
else {
   // do something different
}
```

Java vs. Python – conditionals

```
if (my_var > 2) {
   // do something
}
else if (my_var < 0) {
   // do something else
}
else {
   // do something different
}
```

Java vs. Python – conditionals

```
if (my_var > 2) {
   // do something
}
else if (my_var < 0) {
   // do something else
}
else {
   // do something different
}
```

Curly brackets, { }, show start and end of branch.

Pretty much anything that requires indentation in Python requires curly brackets in Java!

Java vs. Python – conditionals

```
if (my_var > 2) {
   // do something
}
else if (my_var < 0) {
   // do something else
}
else {
   // do something different
}
```

Indentation not technically required…but important for readability

Java vs. Python – comments

```
if (my_var > 2) {
   // do something
}
else if (my_var < 0) {
   // do something else
}
else {
   // do something different
}
```

Single line comments begin with //

Java vs. Python – conditional operators

Python	Java
`and`	`&&`
`or`	`||`
`not`	`!`

Java vs. Python – loops

Python

```
while some_int > 0:
   # Do something
```

Java

```
while (someInt > 0) {
   // Do something
}
```

Note the ( ) and { }

Java vs. Python – loops

Python

```
for elem in arr:
   # Do something
```

Java

```
for (int elem : arr) {
   // Do something
}
```

Note the ( ) and { }

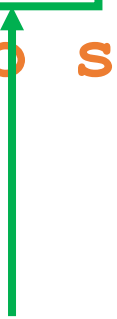
Java vs. Python – loops

Python

```
for elem in arr:
    # Do something
```

Java

```
for (int elem : arr) {
    // Do something
}
```



Specify type of items to iterate

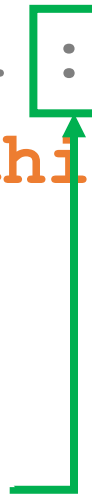
Java vs. Python – loops

Python

```
for elem in arr:
   # Do something
```

Java

```
for (int elem : arr) {
   // Do something
}
```



":" in place of "in"

Java vs. Python – loops

Python

```python
for i in range(10):
    # Do something
```

Java

```java
for (int i = 0; i<10; i++)
{
    // Do something
}
```

Java vs. Python – loops

Python

```python
for i in range(10):
    # Do something
```

Java

```java
for (int i = 0; i<10; i++)
{
    // Do something
}
```

Again, note the ( ) and { }

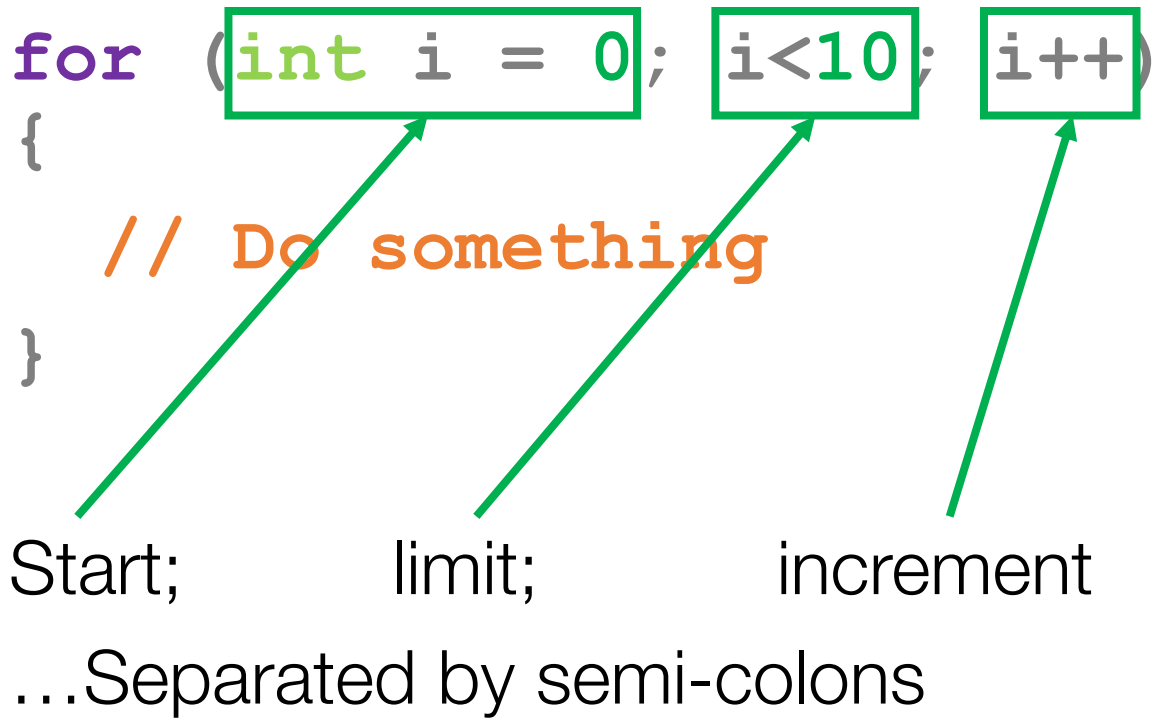
Java vs. Python – loops

Python

```
for i in range(10):
   # Do something
```

Java

```
for (int i = 0; i<10; i++)
{
   // Do something
}
```



…Separated by semi-colons

Some familiar data types in Java

```
int myInt = -3;
float myFloat = 4.67;
String myString = “abc”;
boolean myBool = true;
```

Primitive data types in Java

- **integers**
  - byte e.g. `byte myNum = 0;`
  - short e.g. `short myNum = 0;`
  - int e.g. `int myNum = 0;`
  - long e.g. `long myNum = 0;`
- **floating point numbers**
  - float e.g. `float myNum = 1.3;`
  - double e.g. `double myNum = 1.3;`
- **characters**
  - char e.g. `char myNum =‘a’;`

Primitive data types in Java

- **integers**
  - byte e.g. `byte myNum = 0;`
  - short e.g. `short myNum = 0;`
  - int e.g. `int myNum = 0;`
  - long e.g. `long myNum = 0;`
- **floating point numbers**
  - float e.g. `float myNum = 1.3;`
  - double e.g. `double myNum = 1.3;`
- **characters**
  - char e.g. `char myNum ='a';`

Why multiple integers and floating point data types?

Primitive data types in Java

- **integers**
  - byte e.g. `byte myNum = 0;`
  - short e.g. `short myNum = 0;`
  - int e.g. `int myNum = 0;`
  - long e.g. `long myNum = 0;`
- **floating point numbers**
  - float e.g. `float myNum = 1.3;`
  - double e.g. `double myNum = 1.3;`
- **characters**
  - char e.g. `char myNum ='a';`

Smallest to largest in terms of **memory**

Primitive data types in Java

- **integers**
  - byte e.g. `byte myNum = 0;` min = -128 max = 127
  - short e.g. `short myNum = 0;` min = -32,768 max = 32,767
  - int e.g. `int myNum = 0;` min = -2^{31} max = $2^{31} - 1$
  - long e.g. `long myNum = 0;` min = -2^{63} max = $2^{63} - 1$
- **floating point numbers**
  - float e.g. `float myNum = 1.3;` 6-7 decimal places
  - double e.g. `double myNum = 1.3;` 15-16 decimal places
- **characters**
  - char e.g. `char myNum ='a';`

Introducing arrays

Ordered collection of values, accessible by index.

E.g.

```
["breakfast", "lunch", "dinner"]
```

Java arrays vs. Python Lists

List (Python)

- One list can contain values that have **different data types**
  e.g. [“apples”, 24, True, 1.2]

- **Flexible size** – no need to specify how many elements will be in the list

Array (Java)

- Can only contain a **single type** of data

- **Fixed size** – must specify how many “slots” the array will have

Creating arrays

The basic syntax:

```
<type of data to store>[] <variable name>
   = new <type of data to store>[<# of slots>];
```

E.g.

```
double[] myArray = new double[9];
```

Printing to the command line

Python:

```
print("I'm a message");
```

Java:

```
System.out.println("I'm a message");
```

Intro to Object-Oriented Design

High quality software

…should be:

- correct
- understandable
- modifiable
- efficient

High quality software is **correct**

- “Correct” = Meets **all** specifications / requirements.

- Multiple correct ways to meet specifications…although some approaches are better than others in particular contexts.
  - What makes one approach better than another?
    - Understandability
    - Modifiability
    - Efficiency

- Ensure correctness with **thorough** testing.

High quality software is understandable

- It's not enough to write software that "works"… it also needs to be easy for humans to read and understand.
  - Other engineers
  - You a month or two after you wrote the code

- Good documentation is important!

High quality software is **modifiable**

- Software systems change and evolve – code should be easy to change and evolve
  - By you or others (why it's important to be *understandable…*)

- Design principles, including OOD, help to make code modifiable
  - …but not a guarantee
  - Software design involves anticipating future changes and accommodating them

High quality software is efficient

- Makes good use of system resources – **time** and **space**

- Make your code efficient through careful choice of data types and algorithms
  - The easiest for you to use is not always the best choice

What is object-oriented design?

- A **design paradigm**:
  - An approach/style of developing a software solution

- Other paradigms:
  - Imperative
  - Functional
  - Event-driven
  - and more

Why do we need paradigms?

- Programming is a **design exercise**
- Paradigms help you to design & build high quality software.
  - (All team members follow the same paradigm)

The object-oriented paradigm

- Programs are made up of **objects**
- Objects are defined by **classes**
- Classes contain **fields**
  - Properties, which store data
- ...and **methods**
  - define behavior, help objects to communicate

Classes and objects refresher

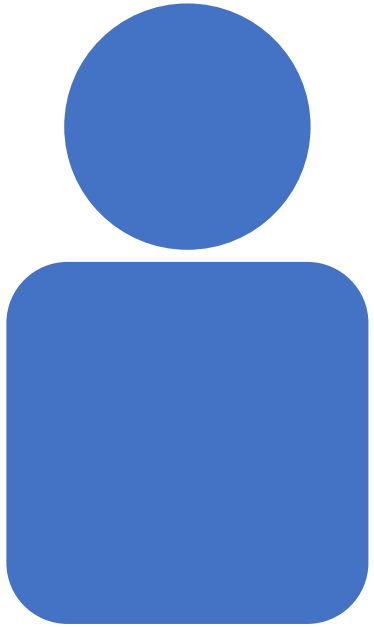
Class

Blueprint for a special datatype, describing it's properties, states and behaviors.

Object

An *instance* of a given class

Example class - Person



Person

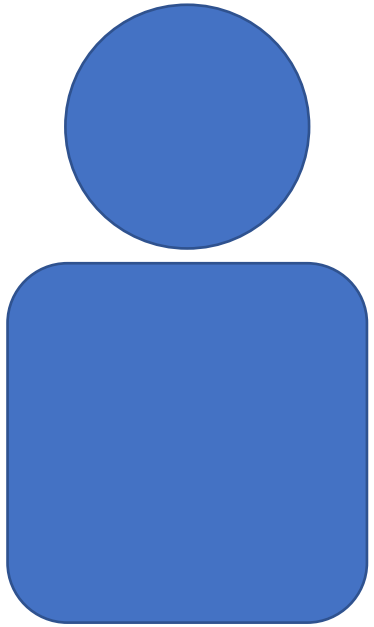
Properties:

- name
- friends

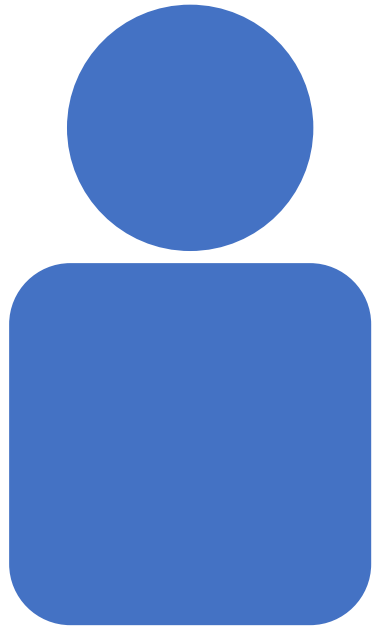
Behavior:

- make friends

Instances of the Person class (objects)

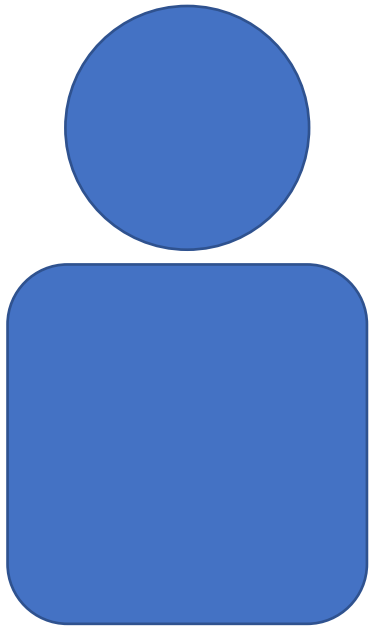


- name = “Luke Skywalker”
- friends = [ ]

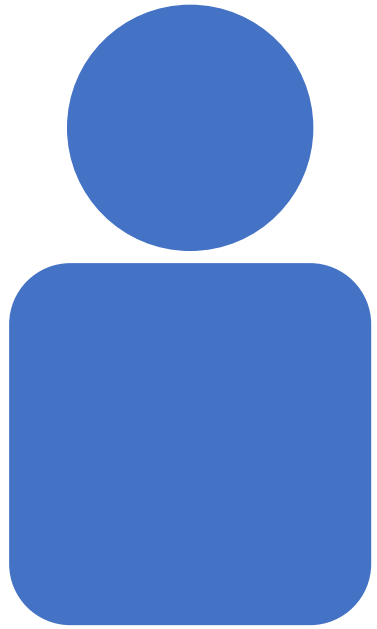


- name = “Princess Leia”
- friends = [ ]

Instances of the Person class (objects)



make friends



- name = “Luke Skywalker”
- friends = [Leia]

- name = “Princess Leia”
- friends = [Luke]

Object-oriented design principles

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

Object-oriented design principles

- **Encapsulation**
  - Keep all the properties and behaviors associated with an object together
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

Object-oriented design principles

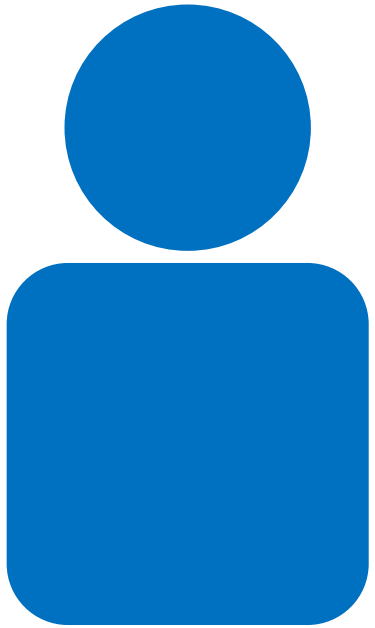
- Encapsulation
- **Abstraction**
  - Keep implementation and “interface” separate
  - Interface – outlines what an object can do but doesn’t actually do it
- Information hiding
- Polymorphism
- Inheritance

Object-oriented design principles

- Encapsulation
- Abstraction
- **Information hiding**
  - Expose only the necessary functionality to users of a class
- Polymorphism
- Inheritance

Information hiding

Student



Properties:

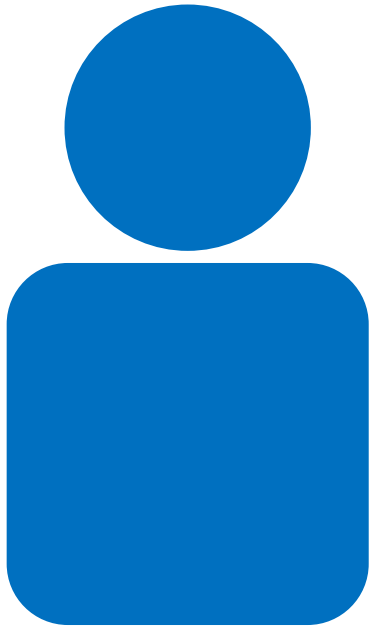
- Courses
- Grades
- GPA

Behavior:

- Calculate and update GPA

Information hiding

Student



Properties:

- Courses
- Grades
- GPA

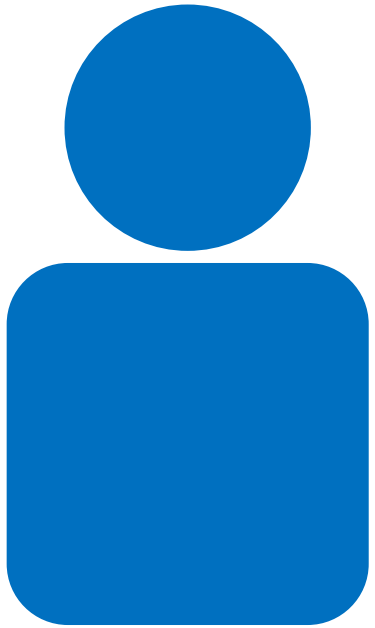
Behavior:

- Calculate and update GPA

Allow users of the Student class to update this…

Information hiding

Student



Properties:

- Courses
- Grades
- GPA

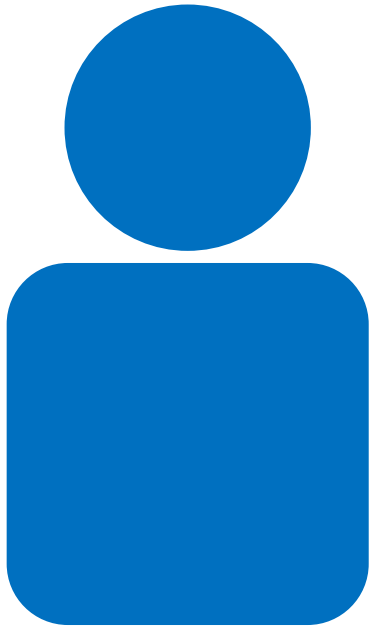
…but not this

Behavior:

- Calculate and update GPA

Information hiding

Student



Properties:

- Courses
- Grades
- GPA

Behavior:

- Calculate and update GPA

Hide this behavior from users

Trigger when a new grade is added.

Object-oriented design principles

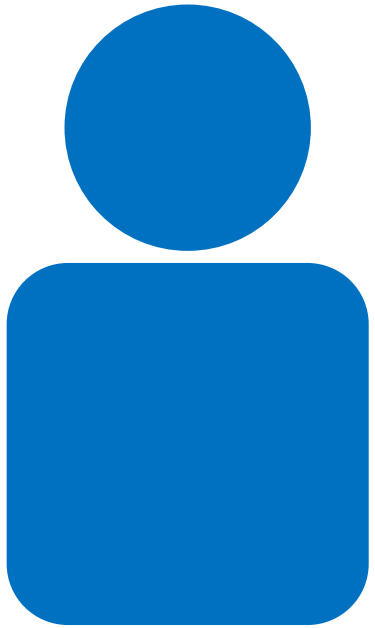
- Encapsulation
- Abstraction
- Information hiding
- **Polymorphism**
  - Objects have the power to act as other objects
  - ... we'll come back to this
- Inheritance

Object-oriented design principles

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- **Inheritance**
  - Classes can extend or override functionality from other classes

Inheritance

Students are people too…



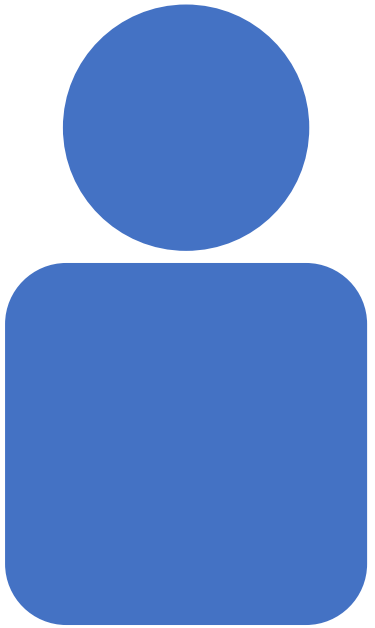
Student

Properties:

- Courses
- Grades
- GPA

Behavior:

- Calculate and update GPA



Person

Properties:

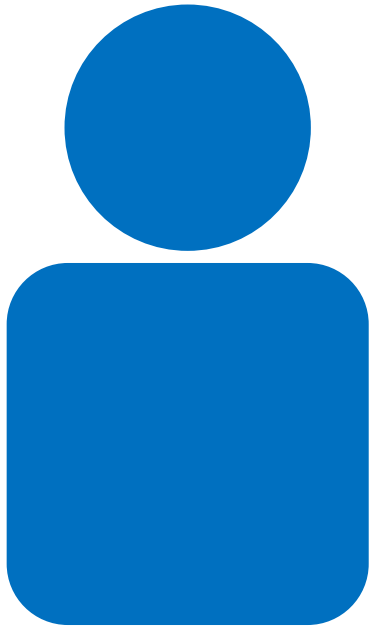
- name
- friends

Behavior:

- make friends

Inheritance

Student **extends** Person



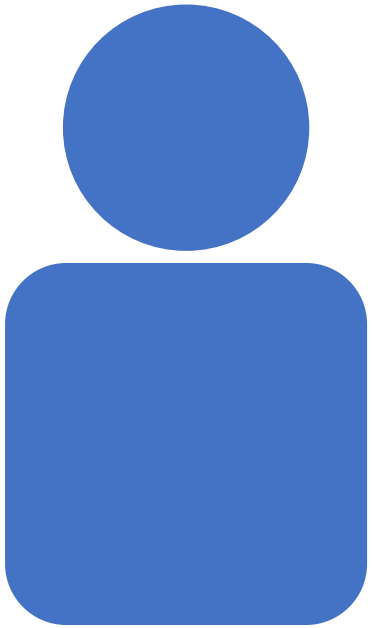
Student

Properties:

- Courses
- Grades
- GPA

Behavior:

- Calculate and update GPA



Person

Properties:

- name
- friends

Behavior:

- make friends

Student **inherits** Person properties and behavior

Classes in Java

EVERYTHING in Java is a class

Classes in Java

- Every class should have a **constructor**
  - Constructor = what happens when an object is created

Example class – Person (Python)

```
def Person:
  def __init__(self, first_name, last_name, yr_of_birth):
    self.first_name = first_name
    self.last_name = last_name
    self.yr_of_birth = yr_of_birth
```

constructor

Example class – Person (Java)

```
class Person {
  private String firstName;
  private String lastName;
  private int yrOfBirth;

  public Person(String firstName, String lastName, int yrOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.yrOfBirth = yrOfBirth;
  }

}
```

constructor

Example class – Person (Java)

```
class Person {
  private String firstName;
  private String lastName;
  private int yrOfBirth;

  public Person(String firstName, String lastName, int yrOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.yrOfBirth = yrOfBirth;
  }

}
```

Constructor name matches class name

Example class – Person (Java)

```
class Person {
  private String firstName;
  private String lastName;
  private int yrOfBirth;

  public Person(String firstName, String lastName, int yrOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.yrOfBirth = yrOfBirth;
  }

}
```

Key difference:

no **`self`** in constructor

Creating an object

Python:

```
j_smith = Person("John", "Smith", 1975)
```

Java:

```
Person jSmith = new Person("John", "Smith", 1975);
```

Stages of creating an object in Java

```
Person jSmith = new Person("John", "Smith", 1975);
```

Declaration – declare a new variable with the given name (jSmith) and the given data type (Person)

Stages of creating an object in Java

```
Person jSmith = new Person("John", "Smith", 1975);
```

Instantiation – actually creates the object with the keyword **`new`**.

Stages of creating an object in Java

```
Person jSmith = new Person("John", "Smith", 1975);
```

Initialization – constructor call initializes the object, passing any initial variables

Classes in Java

- Every class should have a **constructor**
- 3 kinds of variable in a class:
  - Instance variables AKA fields
  - Local variables
  - Class variables

Classes in Java

- Every class should have a **constructor**
- 3 kinds of variable in a class:
  - **Instance variables AKA fields**
    - Belong to an *instance* of the class
    - Created inside the class but outside a method
  - Local variables
  - Class variables

Example class – Person (Java)

```
class Person {
  private String firstName;
  private String lastName;
  private int yrOfBirth;

  public Person(String firstName, String lastName, int yrOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.yrOfBirth = yrOfBirth;
  }

}
```

Instance variables - each Person can have different values for these properties

Example class – Person (Java)

```
class Person {
  private String firstName;
  private String lastName;
  private int yrOfBirth;

  public Person(String firstName, String lastName, int yrOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.yrOfBirth = yrOfBirth;
  }
}
```

Use **`this`** to access instance variables in a method…
just like **`self`** in Python

Classes in Java

- Every class should have a **constructor**
- 3 kinds of variable in a class:
  - Instance variables AKA fields
    - Belong to an *instance* of the class
    - Created inside the class but outside a method
  - **Local variables**
    - Created inside a method and only accessible from that method
    - Destroyed (removed from memory) once the method has executed
  - Class variables

Example class – Person (Java)

```
class Person {
  private String firstName;
  private String lastName;
  private int yrOfBirth;

  public Person(String firstName, String lastName, int yrOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.yrOfBirth = yrOfBirth;
    String fullName = firstName + " " + lastName;
  }
}
```

Local variable – only exists while the method is executing

Classes in Java

- Every class should have a **constructor**
- 3 kinds of variable in a class:
  - Instance variables
    - Belong to an *instance* of the class
    - Created inside the class but outside a method
  - Local variables
    - Created inside a method and only accessible from that method
    - Destroyed (removed from memory) once the method has executed
  - **Class variables**
    - Belong to the *class itself*
    - Created inside the class but outside a method, with the **`static`** keyword

Example class – Person (Java)

```
class Person {
  private String firstName;
  private String lastName;
  private int yrOfBirth;
  private static int minAge;

  public Person(String firstName, String lastName, int yrOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.yrOfBirth = yrOfBirth;
  }
}
```

Class variable - each Person will have the ***same*** value for this property.

Classes in Java

- Every class should have a **constructor**
- 3 kinds of variable in a class:
  - Instance variables
  - Local variables
  - Class variables
- **Use modifiers to control access to variables and methods**
  - Modifiers can do other things too… we'll come back to this later

Classes in Java

- Every class should have a **constructor**
- 3 kinds of variable in a class:
  - Instance variables
  - Local variables
  - Class variables
- Use **modifiers** to control access to variables and methods
  - Access control modifiers:
    - **public**
    - **private**
    - protected
    - <no modifier>

Example class – Person (Java)

```
class Person {
  private String firstName;
  private String lastName;
  private int yrOfBirth;

  public Person(String firstName, String lastName, int yrOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.yrOfBirth = yrOfBirth;
  }

}
```

private = not accessible outside class

e.g.

`<Person object>.firstName` –> error

Example class – Person (Java)

```
class Person {
  private String firstName;
  private String lastName;
  private int yrOfBirth;

  public Person(String firstName, String lastName, int yrOfBirth) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.yrOfBirth = yrOfBirth;
  }

}
```

public = accessible outside class

Constructor must be public

Public or private?

Rules of thumb

- Make fields (e.g. instance variables) **`private`**
- Make constructors **`public`**
- Make methods **`public`** if and only if there is good reason for other code to call a given method.

  Otherwise methods should be **`private`**.

Break (10 mins)

Sample code

Code covered in lectures will be posted on GitHub after class:
https://github.ccs.neu.edu/cs5004-spr21-sea/lecture-code

Getters and setters

Getter = a method to allow users of your class to **get** the value of a private variable (read only)

Setter = a method to allow users of your class to **set** (initialize, change) the value of a private variable

You can choose whether or not your private variables have getters and/or setters

Getters and setters for the Person class

`Person` has three properties:

- **`firstName`**
- **`lastName`**
- **`yearOfBirth`**

Getters and setters for the Person class

`Person` has three properties:

- **`firstName`**
- **`lastName`**
- **`yearOfBirth`**

…Probably useful to allow users to *get* all three properties

Getters and setters for the Person class

`Person` has three properties:

- **`firstName`**
- **`lastName`**
- **`yearOfBirth`**

…Probably useful to allow users to *get* all three properties

…Probably useful to allow users to *set* first and last name

Getters and setters for the Person class

`Person` has three properties:

- **`firstName`**
- **`lastName`**
- **`yearOfBirth`**

…Probably useful to allow users to *get* all three properties

…Probably useful to allow users to *set* first and last name

…Not a good idea to allow users to *set* year of birth

Getters and setters for the Person class

`Person` has three properties:

- **`firstName`** → getter & setter
- **`lastName`** → getter & setter
- **`yearOfBirth`** → getter only

…Probably useful to allow users to *get* all three properties

…Probably useful to allow users to *set* first and last name

…Not a good idea to allow users to *set* year of birth

Getter and setter naming convention

Getter:

```
get<property name or recognizable abbreviation>
e.g. getFirstName
```

Setter:

```
set<property name or recognizable abbreviation>
e.g. setFirstName
```

Typical syntax for a getter

```
public <return type> get<prop name>() {
    return this.<prop name>;
}
```

E.g.:

```
public String getFirstName() {
    return this.firstName;
}
```

Typical syntax for a getter

```
public <return type> get<prop name>() {
    return this.<prop name>;
}
```

public because it needs to be called from outside the class

Typical syntax for a getter

```
public <return type> get<prop name>() {
    return this.<prop name>;
}
```

Must specify the data type of the value to be returned

Typical syntax for a getter

```
public <return type> get<prop name>() {
    return this.<prop name>;
}
```

Don’t forget “this” if you’re working with instance variables

Typical syntax for a setter

```
public void set<prop name>(
                          <param type> <param name>)  {
    this.<prop name> = <param name>;
}
```

E.g.:

```
public void setFirstName(String newName)  {
    this.firstName = newName;
}
```

Typical syntax for a setter

```
public void set<prop name>(
                        <param type> <param name>)  {
    this.<prop name> = <param name>;
}
```

public because it needs to be called from outside the class

Typical syntax for a setter

```
public void set<prop name>(
                          <param type> <param name>)  {
    this.<prop name> = <param name>;
}
```

Must specify the data type of the value to be returned

Use “**`void`**” for setters - nothing is returned

Typical syntax for a setter

```
public void set<prop name>(
                          <param type> <param name>) {
    this.<prop name> = <param name>;
}
```

Pass the new value for the variable as an argument

Typical syntax for a setter

```
public void set<prop name>(
                          <param type> <param name>)  {
    this.<prop name> = <param name>;
}
```

Don’t forget “this”

Why bother?

Why not just make the variables public instead?

Getters and setters:

- help to ensure **correct use** of a class e.g. by performing validation of update values
- help to **simplify** use of a class e.g. performing calculations that don’t need to be made visible to the user

Mutable vs. Immutable objects

In Python:

Lists are **mutable**

```
a_list = ["a", "b", "c"]
a_list.append("c")
```

`a_list` is modified

Mutable vs. Immutable objects

In Python:

Strings are **immutable**

```
hello = "hello"
hello.upper()
print(hello) → prints "hello"
```

`hello` is NOT modified

(Strings in Java are immutable, too)

Writing immutable classes in Java

- Keep instance variables **`private`**
  - Prevents calling code from changing values
- Don’t change the values of instance variables in methods

- If you must make it possible to change a value → *return a new instance of the class*

Is Person mutable or immutable?

```
class Person {
  private String firstName;
  private String lastName;
  private int yearOfBirth;

  public Person(String firstName, String lastName, int yearOfBirth) { ... }

  public String getFirstName() { ... }

  public String getLastName() { ... }

  public int getYearOfBirth() { ... }

  public void setFirstName() { ... }

  public void setLastName() { ... }
}
```

Is Person mutable or immutable?

```
class Person {
  private String firstName;
  private String lastName;
  private int yearOfBirth;

  public Person(String firstName, String lastName, int yearOfBirth) { ... }

  public String getFirstName() { ... }

  public String getLastName() { ... }

  public int getYearOfBirth() { ... }

  public void setFirstName() { ... }

  public void setLastName() { ... }
}
```

Mutable.
Setters change values of instance variable.

An immutable person

```
class Person {
  private String firstName;
  private String lastName;
  private int yearOfBirth;

  public Person(String firstName, String lastName, int yearOfBirth) { ... }

  public String getFirstName() { ... }

  public String getLastName() { ... }

  public int getYearOfBirth() { ... }

}
```

Remove all setters.

... or return a new Object after “changes”

Testing in Java

Testing in Java

- Testing is a **big part** of this course!
  - Last semester provided a very high level into to testing
  - Your tests will be much more thorough and extensive this semester

- We will use a framework called JUnit
  - Tests are written in a separate file from the class to be tested
  - Every method in the class should be tested
  - The following code is a few steps ahead… tomorrow's lab will walk you through creating a test file using JUnit.

Anatomy of a test file

```
<import statements not shown…>
class PersonTest {

  // Add tests here



}
```

All tests for a particular class are encapsulated in a class of their own.

Naming convention:
`<Class to be tested>Test`

Tests are written as *methods* in the test class.

Anatomy of a test file

```
class PersonTest {

  @Before
  public void setUp() {
    // Create Person objects
    // to use in tests
  }



}
```

The first method you write should be used to create example objects of the class to be tested.

Anatomy of a test file

```
class PersonTest {

   @Before
   public void setUp() {
      // Create Person objects
      // to use in tests
   }



}
```

This notation is very important. JUnit will run the method preceded by **`@Before`** before **every test** in the file.

Anatomy of a test file

```
class PersonTest {

  @Before
  public void setUp() {
    // …
  }

  @Test
  public void testOne() {
    // …
  }
}
```

Write tests for every method in your class.

Anatomy of a test file

```
class PersonTest {

   @Before
   public void setUp() {
      // …
   }

   @Test
   public void testOne() {
      // …
   }
}
```

JUnit will run every test (signified by the **`@Test`** notation) in turn, running the **`@Before`** method before each test method.

Fill in some tests

```
class PersonTest {
  private Person amelia;
  @Before
  public void setUp() {
    // …
  }

  @Test
  public void testOne() {
    // …
  }
}
```

Declare example objects as instance variables

Fill in some tests

```
class PersonTest {
  private Person amelia;
  @Before
  public void setUp() {
    amelia = new Person("Amelia", "Earhart", 1897);
  }

  @Test
  public void testOne() {

  }
}
```

Instantiate the example object in the setup method (**`@Before`**)

Fill in some tests

```
class PersonTest {
  private Person amelia;
  @Before
  public void setUp() {
    amelia = new Person("Amelia", "Earhart", 1897);
  }

  @Test
  public void testOne() {
    assertEquals("Amelia", amelia.getFirstName());
  }
}
```

Use **`assertEquals`** to test the values of object properties

Similar to PyTest's assert function

Fill in some tests

```
class PersonTest {
  private Person amelia;
  @Before
  public void setUp() {
    amelia = new Person("Amelia", "Earhart", 1897);
  }

  @Test
  public void testOne() {
    assertEquals("Amelia", amelia.getFirstName());
  }
}
```

The value we **expect** to be returned by the **`getFirstName`** method for the **`amelia`** object.

Fill in some tests

```
class PersonTest {
  private Person amelia;
  @Before
  public void setUp() {
    amelia = new Person("Amelia", "Earhart", 1897);
  }

  @Test
  public void testOne() {
    assertEquals("Amelia", amelia.getFirstName());
  }
}
```

The **actual** value returned by the **`getFirstName`** method for the **`amelia`** object.

What's next?

Todo list:

- Fill in the survey on Canvas!
- Lab tomorrow: environment setup, writing/testing classes

Viewing / reading:

- Align Online videos
  - Same as today but you can pause ☺