GCIS-124
Software Development & Problem Solving

2: Java Classes

RIT | Golisano College of Computing and Information Sciences

SUN	MON (1/22)	TUE	WED (1/24)	THU	FRI (1/26)	SAT
	Unit 1: Python to Java		Unit 2: Java Classes			
	Assignment 1.2 Due (start of class)		Unit 1 Mini Practicum You Are Here Assignment 1.3 Due (start of class)			

SUN	MON (1/29)	TUE	WED (1/31)	THU	FRI (2/2)	SAT
	Unit 2: Java Classes		Unit 3: Inheritance			
	Assignment 2.1 Due (start of class)		Unit 2 Mini Practicum Assignment 2.2 Due (start of class)			

2.0 Accept the Assignment

You will create a new repository at the beginning of every new unit in this course. Accept the GitHub Classroom assignment for this unit and clone the new repository to your computer.

- Your instructor will provide you with a new GitHub classroom invitation for this unit.
- Upon accepting the invitation, you will be provided with the URL to your new repository, click it to verify that your repository has been created.
- You should create a `SoftDevII` directory if you have not done so already.
  - Navigate to your `SoftDevII` directory and clone the new repository there.
- Open your repository in VSCode and make sure that you have a terminal open with the **PROBLEMS** tab visible.

Asking for Help

Could be Improved

I don't understand part 4.c. on the homework.

MUCH Better

Part 4.c. on homework 2.2 asks us to write a function that prompts the user to enter two floating point values and to print the product.

I reviewed slides 17-18 in the lecture, and I finished the practice activity. That all works fine.

But when I try to use the values entered by the user, I am getting an "input mismatch" error. Does anyone have a suggestion?

You have all the resources that you need to solve the problems that have been given to you, but sometimes you may not be able to find the answers to the problems that you encounter.

Asking for help in this course is both ***expected*** and ***encouraged***.

Spending time trying to solve the problem yourself will be a more valuable learning experience in the long run.

If you ***do*** ask for help, try to be as detailed as possible to make it easier for others to help you by providing as much detail as you can.

1. Begin by reviewing the lecture slides related to the problem that you are trying to solve
2. Try to solve the related class activities without looking at the answer
3. When asking for help, try to be as specific as possible about the problem you are having

Java Classes

```java
package lecture;
public class Pet {
    private String name;
    private int age;

    public Pet (String name, int age) {
        this.age = age;
        this.name = name;
    }

    public int getAge () {
        return age;
    }

    public void birthday () {
        age++;
    }
}
```

This week we'll delve more deeping into Java classes and how they compare to Python's

- Last semester we looked at how to create classes and encapsulate state and behavior inside of them using the Python programming language.
- This week we'll explore how to do similar tasks using Java and ways to discuss our design with others.
  - Classes
  - Encapsulation
  - Construction
  - Special Methods
  - Static
  - Class Visualization
  - Potpourri - a mix of useful topics
- Today we will focus on defining and creating instances of classes.

- A typical object-oriented program will contain many small classes that work together to solve a larger problem.
  - But how does the programmer know which classes to create?
  - How do they know what the attributes and methods in each class will be?
- Trying to write a large, complex program without first taking the time to plan will lead to lots of trial and error (wasted time).
  - As well as lots and *lots* of bugs!
- The process of planning which classes, attributes and methods you need *before* you begin writing code is called ***software design***.
- A common technique for starting your design is to perform a ***noun/verb analysis***.
- We start by identifying all of the nouns in the problem statement. Many nouns will become classes or attributes.
  - There is often a system level noun which describes the entire application. Most of the time this noun does not become a class or attribute.
  - Examples: the game, the application, the system
- Many verbs will become methods in one of the classes.

Noun/Verb Analysis

Problem Statement Fragment:

Every item in the Star Wars Battle game has a name. Ships also have a hull rating that determines how much damage it can sustain before destruction. In addition, some ships have shields which recharge over time and must be depleted before the hull takes damage. Each ship also includes some level of armament. When fired, the weapons deal various types of damage to enemy ships.

Most problem statements will contain a very detailed description of the requirements that the software program must satisfy.

2.1 Getting Started With Design

You will soon see that even simple Java programs will often contain many small classes that work together to solve a larger problem. The first step in writing such a program is planning which classes you will need and what the attributes and methods in each class will be. This process is referred to as *software design*. Let's practice designing a software program based on the simple description below.

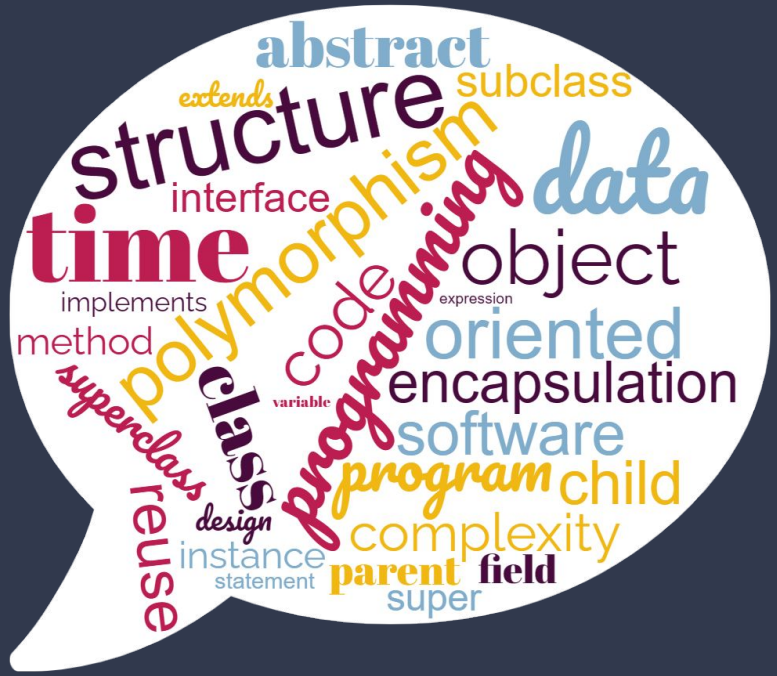
Problem Statement
The Star Wars Battle game simulates space battles between various ships in the Star Wars Cinematic Universe. Every ship has a name and a hull rating that determines how much damage it can sustain before destruction. In addition, some ships have shields which recharge over time and must be depleted before the hull takes damage.

Each ship also includes some level of armament. Larger ships generally have more armament. When fired, the weapons deal various types of damage to enemy ships. Each weapon has a name and there are three types of color coded weapon damage: Heavy (Green), Normal (Red), Ion (Blue).

- Start by creating a new `design.txt` file in the root folder of your repository.
- List all the nouns
  - Indent attributes below their respective class names
- Add the verbs to the listing
  - Indent verbs below their encapsulating class name
  - Include any known parameters inside the methods parentheses

- In Java, almost everything you create will be part of a ***class***.
  - Classes can have ***state*** information.
  - Like everything else it needs a ***visibility*** specifier and a ***type***.
- The only class attributes that are allowed are the ones you ***declare*** (similar to Python slots).
  - The attributes are called ***fields***.
- ***Methods*** are functions that are part of a class.
  - Methods are how we add ***behavior*** to a class
  - Since ***all*** functions in Java must be part of a class, all functions are methods.
- Each method declaration must have:
  - A ***visibility*** specifier (there is a default).
  - A ***return*** type (use void for no return).
  - Types for all parameters.
- From this point forward, you should not use `static` unless you are calling a function directly from `main`.
  - There are other uses of `static` which we will discuss later.

OO Terminology



There are a lot of terms associated with Object-Oriented Programming. Understanding the vernacular can be just as important as programming.

Object Creation

A new instance of a class must be constructed before the object can be used.

- A class is like a ***blueprint***, they tell us how to build something but you cannot use them directly.
  - Think of the blueprint to a house, you cannot live in the blueprint, you must build the house first.
- The process of construction is called ***instantiation***.
  - I.E. You create an ***instance*** (***object***) of a class.
- An ***object*** acts like a ***container*** that holds its ***state*** and ***behavior*** together in one place.
- To invoke instantiation use the `new` keyword along with the class name as a function.
  - Unlike Python, when a Java object is created the space for all of its attributes are ***allocated*** in memory.

2.2 Blasters Ready

When implementing a design, we should start with any classes that do not ***depend*** on any other classes; that means that they do not need any other classes to work. In the Star Wars Battle simulator, the `Weapon` class doesn't depend on any other classes. Start implementing the `Weapon` class based on the analysis that you did earlier.

```
public class House {
  public int numBathrooms;
  public int numBedrooms;
  public boolean hvac;
  public String exteriorColor;

  public static void main(String[] args) {
    House myHouse = new House();
  }
}
```

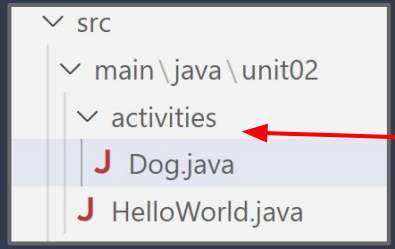
In Java, placing files in a directory adds them to a package (more later). For now, if `package unit02;` was not added automatically to the file, you'll need to manually add it.

- Inside the `src/main/java/unit02` folder, add a new file named "`Weapon.java`".
- Edit the file to include the `Weapon` class and all of its fields.
  - Don't forget to declare the types as well as the names.
  - You can make everything `public` for now, we'll fix it later.
- Create a new `StarWarsBattle` class in the same directory. Add a `main` method which creates an instance of the `Weapon` class.
  - Print out the weapon's `name` and `damage` value.
  - Use the VSCode run button to run the class.

Java Packages

- A ***Java Package*** is a namespace that is used to group related classes together.
- Packages are used to better organize your code so that you do not end up with dozens (or hundreds) of classes in the same folder in a large software project.
- The ***default package*** is in the `src/main/java` folder in a Maven project and it *does not have a name*.
  - Classes in the default package do not include a package statement at the top.
  - Use of the default package is strongly discouraged. In fact, Maven will generate an error if you try to put a class in the default package!
- For other packages, there is a very strong correlation between the name of a package and its location in the file system.
  - These packages are in subfolders, e.g. `src/main/java/my/package/name`
  - The name of the package must match the folder(s) exactly, including case.
  - If the folders are nested, dots (`.`) are used to separate the names, e.g. `package my.package.name;`

Classes in a package are located in a folder with the same name as the package.

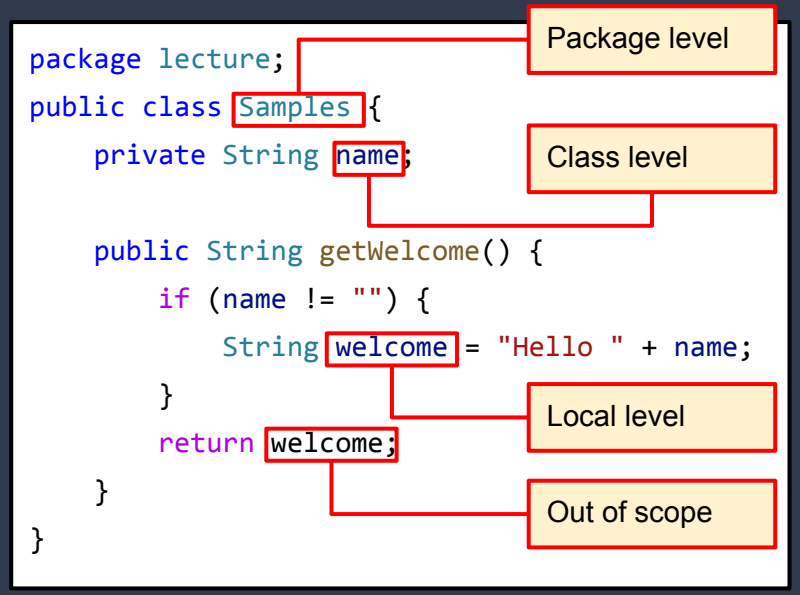


The `Dog` class is in the `unit02.activities` package in the `src/main/java/unit02/activities` folder in the project.

The class must also include a corresponding ***package statement*** at the top of the class.

```
package unit02.activities;

public class Dog {
}
```

Packages & Scope

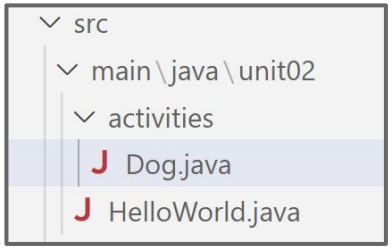


In addition to the package, class, and function levels, any set of curly braces (`{}`) also creates a new scope for any variables declared inside.

- ***Scope*** refers to the parts of a program in which an identifier is ***visible***.
  - The identifier may refer to the name of a variable, a class, or a method.
- There are several different types of scope in Java.
  - The scope of a ***local variable*** is the block of code in which it is defined, e.g. a variable that is declared inside of curly braces (`{}`) is not visible from outside of those braces.
  - The scope of a ***field*** is the entire class in which it is declared, i.e. it can be used in any method in the class.
  - The scope of a ***method*** is also the entire class in which is is declared.
- Java also includes ***package scope***.
  - Classes declared within the same package are visible to each other without using an `import` statement.
  - Classes outside of the package must use an `import` statement to access the classes inside the package.
  - This is why an `import` statement is needed to use the `Scanner` or File I/O classes.
- Classes in the `java.lang` package are special and do not need to be imported.
  - This is why you did not need to import the `System` or `String` classes to use them.

2.3 Java Packages

Large software projects will include dozens (if not hundreds) of small classes. Java packages are used to place related classes into the same namespace. Conversely, unrelated classes are separated into different namespaces to keep everything better organized. Let's move the Star Wars Battle classes that we have written so far into a new package.




```
package unit02.activities;

public class Dog {
}
```

The package statement must exactly match the folder name(s).

- Create a new folder named "`swb`" (short for Star Wars Battle) under the `src/main/java/unit02` folder in your project.
- Click and drag the `Weapon.java` file from the `unit02` folder into the `swb` folder.
- Open the `Weapon` class and verify that VSCode has added the correct package statement to the top of the class.
  - It should be `package unit02.swb;`
  - If the package statement is missing, add it manually.
- Repeat the process for the `StarWarsBattle` class.
- Run the `main` method in `StarWarsBattle` to make sure that it is still working correctly.

Constructors

- As we learned with Python, the special function used to create a new instance of a class is called a ***constructor***.
  - It is a method that has the same name as the class.
- Whenever you declare a class, a special construction method is automatically added to it.
  - This method is known as the ***default constructor***.
  - It does not declare any parameters.
  - When called (using the `new` operator) a new instance of the class is created.
- When the default constructor is used, Java assigns default values to ***all*** of the attributes defined by the class.
  - `0` for numerics.
  - `false` for Booleans.
  - `null` for objects.
- This is a significant difference between Python and Java: ***all*** Java objects are guaranteed to assign values to any fields defined by the class whether you write a constructor or not.

Java provides a default constructor for all classes.

```
public class House {
    public int numBathrooms;
    public int numBedrooms;
    public boolean hvac;
    public String exteriorColor;

    public static void main(String[] args) {
        House myHouse = new House();
    }
}
```

Default constructor which creates a House instance

Any fields defined by the class will be assigned default values in the newly created object.

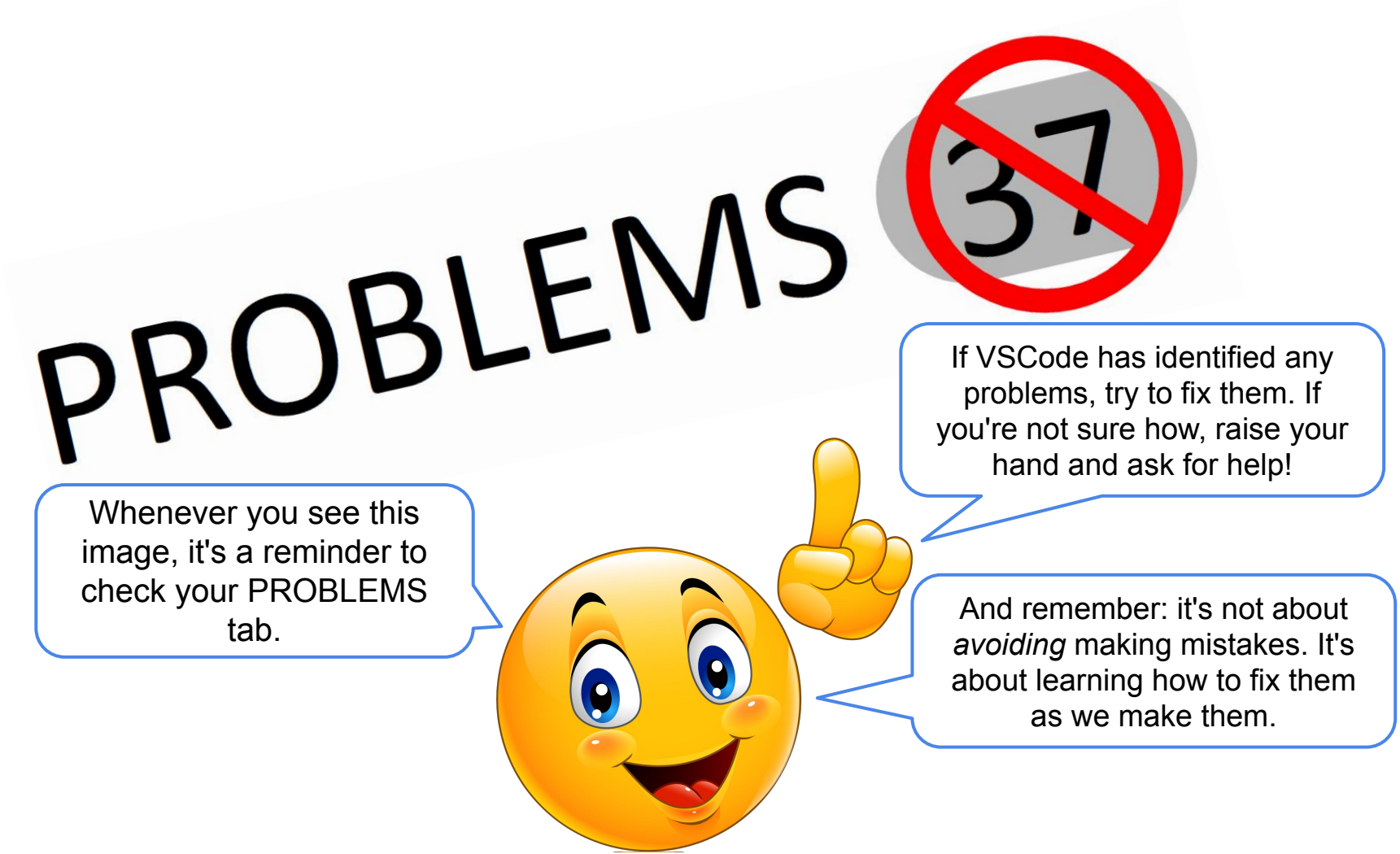
- While the default constructor is useful, it only assigns default values to the fields in an object when it is created.
  - If we want to assign some other value to each field, we will need to do in manually each time an object is created.
- In Python, we solved this problem by writing our own ***initializing constructor***.
  - We used parameters to the constructor to initialize the fields in the new object.
- In Java, an initializing constructor should have the following properties:
  - It should have ***public*** visibility.
  - It should declare ***no*** return value (not even void).
  - It should declare a ***parameter*** for each field in the class.
- The keyword `this` is used to reference the ***instance*** of the class from within itself.
  - It is similar to ***self*** in python.
- In most circumstances it is not needed ***explicitly***.
  - You can use `this` for ***disambiguating*** and between fields and variables with the same name.

this VS self

It is considered good programming style to name constructor parameters the same as the fields that they are being used to initialize.

```
public class Pet {
    public String name;
    public int age;
    public String species;

    public Pet(String name, int age,
               String species) {
        this.age = age;
        this.name = name;
        this.species = species;
    }
}
```

`this` is used to differentiate between the fields in the new object and the parameters with the same name.


PROBLEMS 37
If VSCode has identified any problems, try to fix them. If you're not sure how, raise your hand and ask for help!
Whenever you see this image, it's a reminder to check your PROBLEMS tab.
And remember: it's not about *avoiding* making mistakes. It's about learning how to fix them as we make them.

2.4 Initializing Constructors

PROBLEMS 37

Java's default constructor will assign default values to any fields defined by the class when a new object is created which avoids a lot of the problems that we ran into when working with Python classes and objects. Unfortunately, the default values are not always very useful. Let's add an initializing constructor to the Weapon class that declares parameters for each of the fields in the class.

```
public class Pet {
    public String name;
    public int age;
    public String species;

    public Pet(String name, int age,
               Species species) {
        this.age = age;
        this.name = name;
        this.species = species;
    }
}
```

- Open the `Weapon` class and define a new constructor.
  - It should have the same name as the class.
  - It should declare a parameter for each field. The parameters should be named the same as the field that they are initializing!
  - It should not declare a return type.
- What happens when you try to run the `StarWarsBattle` class now?
  - If any errors have occurred, do you know *why*?
  - Can you fix them? If not, raise your hand!

2.5 Riding in Ships with Moisture Farmers

Even small Java programs will need more than one class to meet all of the requirements. Let's start on the next class in the Star Wars Battle game. Use your noun/verb analysis to create the `Ship` class along with an initializing constructor.

```
public class Pet {
  public String name;
  public int age;
  public String species;

  public Pet(String name, int age,
             String species) {
    this.age = age;
    this.name = name;
    this.species = species;
  }
}
```

- Add a new `Ship.java` file to the `swb` package
- Define the class along including all of its state
  - Refer back to the noun-verb analysis to find the state information
  - Once again, make all the attributes public
- Add a constructor to the class that accepts all of the state information as parameters
- Inside the `main` method (in `StarWarsBattle`) call the default constructor to create a new `Ship`
  - I.E. `Ship myShip = new Ship()`
  - What happens?
- Modify `main` to call the constructor that was added to the class
- Print the ship's `name` and `hullAmount`

- Unlike Python, Java ***does not*** have default parameter values. Instead, Java allows two or more methods to have the same name as long as the ***parameter lists*** are different.
  - Having two methods with the same name in the same class is referred to as ***overloading***.
- Constructors are just a special kind of method with some extra rules.
  - Must be named the same as the ***class***.
  - Do not have a ***return*** type.
- Therefore, constructors can be overloaded just like any other method.
  - We can call one constructor from another and use default values as arguments to the other constructor.
  - One constructor calls another using this, e.g. `this("Hermione", 2, "Basset Hound")` .
  - I `this` is used to call another constructor it must be the first line of the constructor!
- If you write any constructor you ***lose*** the default one that Java provides.
  - You can replace the removed default constructor with your own ***parameterless*** constructor.
- Simply write a constructor that doesn't declare any ***parameters*** that calls another constructor with default arguments.

Overloading & Constructors

```
1  public House(String color, int baths,
2               int beds, boolean hvac) {
3      exteriorColor = color;
4      numBathrooms = baths;
5      numBedrooms = beds;
6      this.hvac = hvac;
7  }
8
9  public House(String color) {
10     this(color, 2, 3, true);
11 }
```

Notice the use of `this` instead of `House` when calling a constructor from inside a constructor.

This example will make a house of any color, 2 bathrooms, 3 bedrooms, and HVAC installed.

2.6 Overloading Constructors

Overloading constructors can provide different (and often simpler) ways to construct different instances of the same class. Consider that some ships may not have weapons or shields. Overload the `Ship`'s constructor to make it easier to construct such a ship.

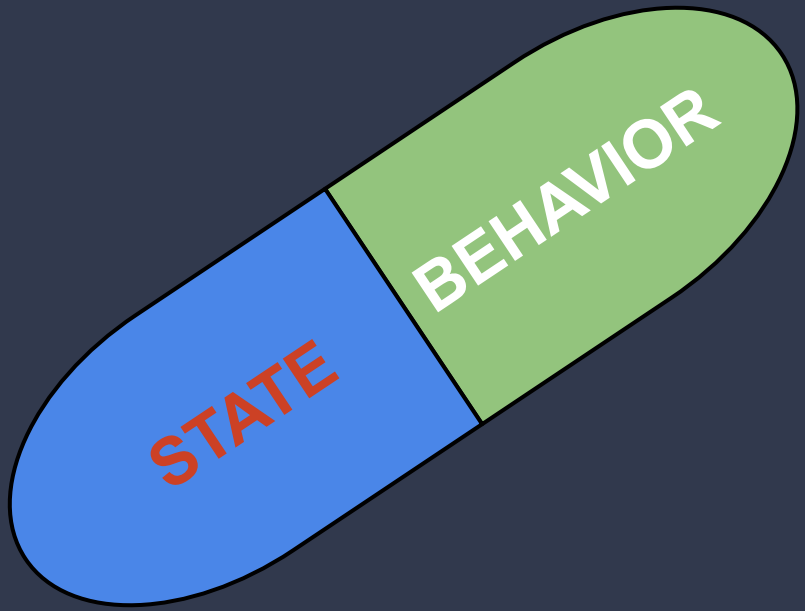
```
public House(String color, int baths,
             int beds, boolean hvac) {
    exteriorColor = color;
    numBathrooms = baths;
    numBedrooms = beds;
    this.hvac = hvac;
}

public House(String color) {
    this(color, 2, 3, true);
}
```

When one constructor calls another, this is referred to as ***constructor chaining***.

- Add a second constructor to the the `Ship` class that only declares parameters `name` and `hullAmount`.
  - Make sure to use `this` to call the other constructor using default values for the other parameters. What values should you use by default?
- In `main`, create a new `Ship` using the simplified constructor and print the new ship's `name` and `hullAmount`.
  - What happens when you print the other attributes of the new ship?

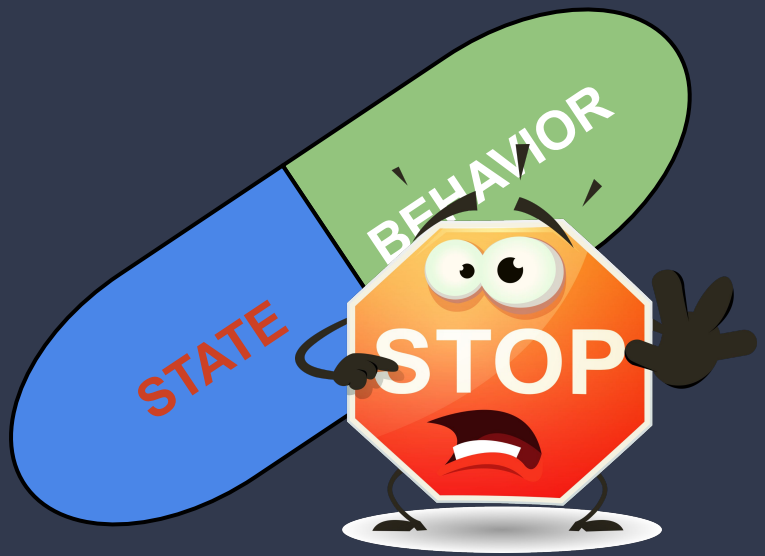
- Now that we know about classes, you might ask "Why are they so popular?"
- The main reason they have become the dominant form of programming is ***encapsulation***.
  - Encapsulation refers to putting something ***inside*** something else, e.g. putting related ***fields*** and ***methods*** together inside of a class.
  - Each object of the class that is created has its own copy of the fields and methods.
  - Changes to the state of one object do not have any effect on the state of any other objects.
- Encapsulation allows grouping of related ***state*** (fields) and ***behavior*** (methods) together.
  - The fields in the class should be related to each other and collectively represent something.
  - The methods in a class should use one or more of the fields in some way.
- Encapsulation also allows us to easily ***reuse*** our classes since the state and behavior is co-located.
  - Classes and objects can be used in lots of different places within the program.

Encapsulation



Encapsulation refers to the fact that an object acts like a container that holds its ***state*** and ***behavior*** together as one, cohesive unit.

Visibility



In addition to containing related state and behavior, another key aspect of encapsulation is ***data privacy***.

Objects prevent other parts of the program from accessing or modifying their state.

- The other major reason we encapsulate state and behavior is to protect and control ***access*** to it.
  - We want to be able to enforce the rules regarding the way that state is changed, e.g. we don't want a ship's shields to fall below 0.
  - We do not want the fields inside an object to be accessed or modified by other parts of the program.
- To do this in Java, we use ***access modifiers*** to change the ***visibility*** of methods and fields.
  - `public` – grants access to all parts of the program.
  - `protected` – grants access to classes in the same package and child classes (we'll talk about this more in the next unit).
  - `private` – grants access to objects of the same class.
  - ***<none>*** - grants access to other classes in the same package but *not* child classes. Also known as "package private."
- In general, state (fields) should be ***private*** and accessed or modified *only as needed* using ***public*** methods.
- Protecting fields from being manipulated by other parts of the program is referred to as ***data privacy***.
  - This is a core concept in object-oriented programming.

2.7 Controlling Access to State

We do not want other parts of the program to modify the fields inside of our objects (either accidentally or on purpose). The mechanism for preventing access is to use access modifiers for all fields. Let's modify our classes to make all of the fields `private`.

```
public class Samples {
   private String name;

   public String getWelcome() {
      String welcome = "Hello ";
      if (name != "") {
         welcome = welcome + name;
      }
      return welcome;
   }
}
```

- Open the `Weapon` class and change the access modifiers for all of the fields to make them private.
- What happens when you try to run the `StarWarsBattle`class?
  - If there are any errors, do you know why?
- Repeat this process with the `Ship` class.
  - There may be more errors!
  - How should we fix them?

- Because all the data is private there is no way to access the state directly from another class.
  - This will cause errors in any code that tries to access the fields directly.
- If other parts of the program *must* ***access*** the private data we use ***accessors***.
  - An accessor is named using the "`get`" prefix and the name of the field, e.g. `getName`.
  - An accessor typically contains only a return statement that returns the value of the field.
  - Accessors are also called "getters."
- If other parts of the program *must* ***change*** the private data, we use ***mutators***.
  - A mutator is named using the "`set`" prefix and the name of the field, e.g. `setName`.
  - A mutator declares one parameter and uses it to modify the value of the field with the same name.
  - Mutators are also called "setters."
- Accessors and mutators should ***only*** be written if they are ***needed*** by other parts of the program.

Accessors & Mutators

Getters and setters allow other parts of the program to access and/or modify the private data inside of an object.

```
1  public class Pet {
2      private String name;
3      private int age;
4
5      public int getAge() { return age; }
6
7      public void setName(String name) {
8          this.name = name;
9      }
10 }
```

This ***breaks encapsulation*** because it exposes private data outside of the class. Getters and especially setters should only be added if *absolutely necessary*.

2.8 Accessing & Mutating

Using proper encapsulation means that an object protects and controls access to is fields by using access modifiers to make them private. If the values of the fields are needed outside of the class, you will need to add accessors and/or modifiers to enable that access.

```
1  public class Pet {
2      private String name;
3      private int age;
4
5      public int getAge() { return age; }
6
7      public void setName(String name) {
8          this.name = name;
9      }
10 }
```

It is almost never a good idea to make a field public. The coding standards for this class require that all non-static fields be private.

- Update the `Weapon` class to include accessors only as needed.
  - Only add accessors where at least one other part of the program needs to be able to see the field.
  - ***Hint***: add them to fix any errors cause by making the fields private.
  - You can always add more accessors later as needed!
- Update the `Weapon` class to include mutators only as needed.
  - An object may change the value of one of its own fields. That does not mean that a mutator is needed. Ask yourself "Will some *other part* of the program need to *directly* set the value of the field?"
  - If the answer is **no**, do not create a mutator for that value.
  - If it is, **yes**, then add the mutator.
  - If the answer is **maybe**, delay creating a mutator until the implementation specifically requires it.
- Repeat the entire process for the fields in the `Ship` class.
- Run the `StarWarsBattle` class to make sure that everything is working as expected.

Enumerated Types (`enum`)

A new enumerated type is declared as an enum rather than a class.

```
1  public enum Species {
2      CAT,
3      DOG,
4      BIRD,
5      FISH,
6      LIZARD,
7      HAMSTER
8  }
```

The values that may be assigned to a variable of the enumerated type are explicitly specified as a comma-separated list of names.

```
Species species = Species.DOG;
```

- Sometimes a variable should only be assigned one of a specific set of predefined values, e.g.:
  - Days of the week: Monday, Tuesday, etc.
  - Months of the year: January, February, etc.
  - Species of pet: cat, dog, hamster, etc.
- What Java type would you use to represent a variable like this?
  - You could use an `int`, e.g. 0=Monday, 1=Tuesday, and so on. But what would stop you from assigning a value of -1 or 327 to the variable?
  - You might try a `String`, but you'd have the same problem: you could assign *any* string to the variable.
- In Java, it is possible to create an ***enumerated type*** (or an `enum` for short).
  - The values that may be assigned to the type are explicitly enumerated using a comma-separated list of names.
- Once created, an enum can be used like any other type as variables, parameters, etc.
  - ***Only*** values from the enumerated list may be assigned to a variable of the enum's type.

2.9 A Damage Type `enum`

We know that the weapons in the Star Wars Battle game have one of three different damage types: *heavy*, *normal*, and *ion*. We'd like to restrict the damage type so that if can only be one of that specific set of predefined values. Let's create an enum to represent the damage type!

```
public enum Species {
    CAT,
    DOG,
    BIRD,
    FISH,
    LIZARD,
    HAMSTER
}
```

- Create a new file in the `swb` package named "`DamageType.java`".
  - Use the example to the left to create an enumerated type that lists the specific, pre-defined values that may be assigned to a `DamageType` variable: *heavy*, *normal*, and *ion*.
  - By stylistic convention, the identifiers should be UPPERCASE.
- Update the `Weapon` class to use your new enumerated type for the damage type.
  - Replace all instances of the old type with the new `DamageType` including fields, parameters, return types, etc.
- Fix any errors that occur in the `StarWarsBattle` class.
  - Run the `main` method to make sure that everything is working as expected.

Complex `enum`s

```
1  public enum BoilingPoint {
2      CELSIUS('C', 100),
3      FAHRENHEIT('F', 212),
4      KELVIN('K', 373.5f);
5
6      private char scale;
7      private float degrees;
8
9      private BoilingPoint(char scale,
10                          float degrees) {
11         this.scale = scale;
12         this.degrees = degrees;
13     }
14
15     public char getScale() {
16         return scale;
17     }
18
19     public float getDegrees() {
20         return degrees;
21     }
22 }
```

There's a lot going on here! Let's take a closer look…

- Sometimes we would like to associate one or more additional values with each of the predefined values in an enumerated type.
- It is possible to do this by declaring ***fields*** inside of an `enum`.
  - Each of the predefined values in the `enum` should assign a value to each of the fields.
  - This is done by defining a ***constructor*** in the `enum` and using it to initialize the variables.
  - The constructor may declare ***parameters***, in which case arguments must be passed into the constructor when each of the predefined values is declared.
- An `enum` may also define methods.
  - It is common to write an accessor for each of the fields.
  - You may add other methods as well.
- Adding these features to an `enum` significantly complicates the syntax.
  - The syntax resembles a full-blown Java class, but the `enum` still retains the benefit of only being assignable to a specific list of predefined values.

A Closer Look at Complex Enumerated Types

```
1  public enum BoilingPoint {
2      CELSIUS('C', 100),
3      FAHRENHEIT('F', 212),
4      KELVIN('K', 373.5f);
5
6      private char scale;
7      private float degrees;
8
9      private BoilingPoint(char scale,
10                          float degrees) {
11         this.scale = scale;
12         this.degrees = degrees;
13     }
14
15     public char getScale() {
16         return scale;
17     }
18
19     public float getDegrees() {
20         return degrees;
21     }
22 }
```

A Closer Look at Complex Enumerated Types

The enum may declare one or more ***fields***. If the fields are not explicitly initialized, Java will assign default values.

It is common to write ***accessors*** for each of the fields so that they can be accessed from outside of the enumeration.

```
1  public enum BoilingPoint {
2      CELSIUS('C', 100),
3      FAHRENHEIT('F', 212),
4      KELVIN('K', 373.5f);
5
6      private char scale;
7      private float degrees;
8
9      private BoilingPoint(char scale,
10                          float degrees) {
11         this.scale = scale;
12         this.degrees = degrees;
13     }
14
15     public char getScale() {
16         return scale;
17     }
18
19     public float getDegrees() {
20         return degrees;
21     }
22 }
```

A `private` ***constructor*** may be defined to explicitly initialize any fields. ***Arguments*** must be provided for each ***parameter*** when the predefined values are declared.

2.10 Upgrading `DamageType`

Each of the different types of damage in the Star Wars Battle game has an associated color: green (heavy), red (normal), and blue (ion). Let's upgrade the `DamageType enum` so that it associates the appropriate color with each of the predefined values.

```
public enum BoilingPoint {
    CELSIUS('C', 100),
    FAHRENHEIT('F', 212),
    KELVIN('K', 373.5f);

    private char scale;
    private float degrees;

    private BoilingPoint(char scale,
                         float degrees) {
        this.scale = scale;
        this.degrees = degrees;
    }

    public char getScale() {
        return scale;
    }

    public float getDegrees() {
        return degrees;
    }
}
```

- Open the `DamageType enum` and upgrade it so that the appropriate color is associated with each of the predefined damage types.
  - Declare a `private` field of an appropriate Java type.
  - Define a `private` constructor that declares a parameter for the `color` and use it to initialize the field.
  - Modify the declaration of each of the predefined values so that it provides an appropriate argument to the constructor.
  - Add an accessor for the color.
- Update the `main` method in the `StarWarsBattle` class to print the color of each weapon's damage type.

Special Method: `toString()`

- All classes contain a `toString()` method with the definition of:

  `public String toString() {...}`
- If you do not replace the default implementation, it will return the name of the ***object*** (plus some other *stuff* that we will talk about later), e.g.:

  `unit02.activities.Pet@2328c243`
- The `toString()` method is automatically called on an object whenever you:
  - Attempt to ***print*** the object to standard output.
  - Attempt to ***concatenate*** the object onto a string with the `+` operator, e.g. `"Pet: " + myPet`
- Thus it is often beneficial to write your own version that returns something more useful than the default string.
  - Replacing an existing method with your own implementation is called ***overriding***.
  - When overriding, you should use the `@Override` annotation.

```
1  public class Pet {
2      private String name;
3      private int age;
4      private Species species;
5
6      @Override
7      public String toString () {
8          return name + " is a " +
9              age + " year old " +
10             species + ".";
11     }
12 }
```

The `toString()` special method is used to control how a class is translated into a string.

2.11 Overriding `toString()`

If we try to print one of our `Weapon` objects to standard output, the string that is printed will look something like `Weapon@12a3db47`. That is not particularly useful. However, if we override the `toString()` method we can replace the default string representation of an object with our own version. Let's try it now!

```
public class Pet {
    private String name;
    private int age;
    private Species species;

    @Override
    public String toString () {
        return name + " is a " +
            age + " year old " +
            species + ".";
    }
}
```

The `@Override` annotation is not required, but it can be very helpful for a number of reasons and it is considered good practice to use it.

- Open the `Ship` class and override the `toString` method for the `Ship` to return a string in the format:
  - `"<name>: shields <shields>, hull <hullAmount>"`
    e.g. `"X-Wing: shields 50, hull 12"`
- Repeat this process for the `Weapon` class.
  - You may choose a format for the string that is returned, but you should try to include all of the relevant details.
- In the `StarWarsBattlemain`, print one of the ships and one of the weapons that you created previously to standard output.

Special Method: `equals(Object)`

```
1  @Override
2  public boolean equals(Object obj) {
3      if(obj instanceof Pet) {
4          Pet other = (Pet)obj;
5          return (age == other.age) &&
6              name.equals(other.name);
7      } else {
8          return false;
9      }
10 }
```

Remember that two instances of the same class can directly access each other's private fields without using an accessor.

- Like Python, Java includes two different methods for comparing values for equality.
- The == operator is used in a boolean expression to compare two values for ***shallow equality***.
  - e.g. `x == y`
  - When comparing primitive types, the expression will be true if the ***values*** are the same.
  - When comparing ***objects***, the expression will be true if the ***addresses*** are the same. This is only true if x and y are the same object.
- The `equals(Object)` method is used to compare two objects for ***deep equality***.
  - e.g. `x.equals(y)`
  - The method returns true if the x and y are equivalent to each other.
  - The default implementation provided by Java works just like the == operator - it returns `true` if `x` and `y` are the same object.
  - The method must be overridden to change this.
- In general you will want to use `equals` instead of == when working with non-primitive objects.
  - Even though == will appear to work sometimes with strings, you should generally use `equals` when comparing two ***strings***.

A Closer Look at the `equals(Object)` Method

```
1  @Override
2  public boolean equals(Object obj) {
3     if(obj instanceof Pet) {
4        Pet other = (Pet)obj;
5        return (age == other.age) &&
6           name.equals(other.name);
7     } else {
8       return false;
9     }
10 }
```

A Closer Look at the `equals(Object)` Method

The `@Override` annotation is not required, but is recommended.

Not all objects have an `age` or a `name` like `Pet`s do. In order to access those fields, we will need to ***cast*** `obj` into a `Pet` variable, e.g. `Pet other = (Pet)obj;`

If the `Object` parameter is not an instance of the correct class, return `false`.

```
1  @Override
2  public boolean equals(Object obj) {
3      if(obj instanceof Pet) {
4          Pet other = (Pet)obj;
5          return (age == other.age) &&
6              name.equals(other.name);
7      } else {
8          return false;
9      }
10 }
```

Any kind of Java object may be passed into the method, so it is important that we make sure it is the right type using `instanceof`.

The `==` operator should be used when comparing primitive values, and `equals(Object)` should be used to compare reference types like strings.

It is up to the programmer to decide what it means for two instances two be equal to each other. In this case, two `Pet`s are equal if they have the same values for `age` and `name`.

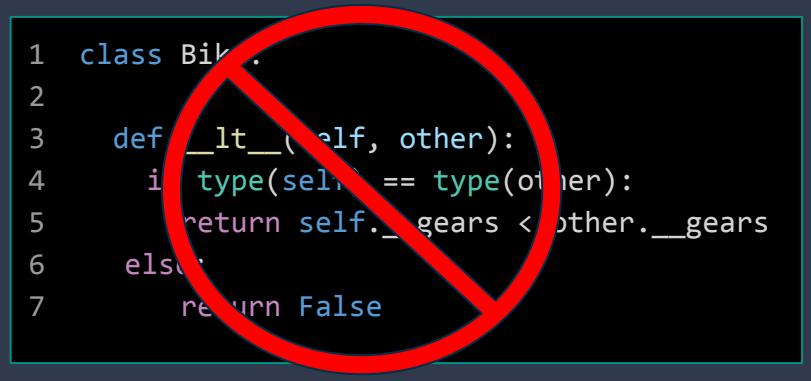
2.12 A Blaster By Any Other Name

The == operator will only evaluate to `true` if an object is compared to itself, but we'd like two `Weapon`s to be considered equal if they have the same damage amount and damage type.

```
@Override
public boolean equals(Object obj) {
   if(obj instanceof Pet) {
      Pet other = (Pet)obj;
      return (age == other.age) &&
         name.equals(other.name);
   } else {
     return false;
   }
}
```

- Open the `Weapon` class and override the special `equals(Object)` method.
  - Use the `@Override` annotation and make sure that the method signature matches the example to the left.
  - Use `instanceof` to make sure that the `obj` parameter is another `Weapon`. Return `false` if it is not.
  - Don't forget to cast the `obj` parameter into a `Weapon`!
  - Return `true` if both weapons have the same damage amount and damage type.
- Open the `main` method in the `StarWarsBattle`class and create at least three `Weapon` objects.
  - Two of the `Weapon`s should have the same damage type and amount, and the third weapon should be different in some way.
  - Print the results of comparing the weapons to each other using both == and `equals(Object)` .

What About the Other Special Methods?



For the purposes of sorting, Java manages object comparisons using a different strategy than Python. This will be discussed in detail in a later unit.

- ***Operator overloading*** refers to replacing the default functionality of an operator.
- In Python, there are several other special methods that may be implemented to overload comparison operators:
  - `__lt__` overloads less than (<)
  - `__lte__` overloads less than or equal to (<=)
  - `__gt__` overloads greater than (>)
  - `__gte__` overloads greater than or equal to (>=)
- Java does not support operator overloading, and so there is no way to replace the default behavior of any of the operators in Java.
- For the purposes of sorting, Java manages object comparisons in a different way.
  - We'll talk about how this works in the ***data structures*** units.
- Python's special `__hash__` method is implemented so that objects will work with hashing data structures like sets and dictionaries.
  - Java features an equivalent method that we will discuss in more detail in the data structures units.

More About `enum`s

- A Java `enum` is really a full-blown class with some extra requirements and features.
  - We already know that we can add fields, constructors, methods, and so on.
  - Only the values enumerated inside of the enum can be assigned to a variable of the `enum`'s type.
- Java `enum`s also have quite a few other features.
- The default `toString()` method returns the name of the value.
  - e.g. `BoilingPoint.KELVIN` will return `"KELVIN"`.
- The `name()` method will also return the name of a value, similar to `toString()`.
  - e.g. `BoilingPoint.CELSIUS.name()` will return `"CELSIUS"`.
  - It can be used inside the methods of the enum.
- The `valueOf(String)` method returns the value with the name that matches the string.
  - e.g. `BoilingPoint.valueOf("CELSIUS")` will return `BoilingPoint.CELSIUS`.
  - Specifying the name of a value that does not exist will throw an `IllegalArgumentException`.
- The `values()` method will return an array of the enumerated values.
  - e.g. `BoilingPoint.values()` will return an array containing the `FAHRENHEIT`, `CELSIUS`, and `KELVIN` values.

The `toString()` and `valueOf(String)` methods can be used to easily convert values of an `enum` to-and-from strings.

```
1   String kelvin = BoilingPoint.KELVIN.toString();
2   System.out.println(kelvin);
3
4   BoilingPoint celsius =
5       BoilingPoint.valueOf("CELSIUS");
6   System.out.println(celsius);
7   BoilingPoint error = BoilingPoint.valueOf("no");
8
9   BoilingPoint[] values = BoilingPoint.values();
10  System.out.println(
11      Arrays.toString(values));
```

The `values()` method will return an array that contains ***all*** of the values enumerated in the `enum`.

2.13 More `enum` Features

Java enums are really full-blown classes with lots of special features. Practice using some of the methods that convert values of an enumerated type to-and-from strings. In addition, print an array of all of the values in an enumerated type.

```
String kelvin = BoilingPoint.KELVIN.toString();
System.out.println(kelvin);

BoilingPoint celsius =
                BoilingPoint.valueOf("CELSIUS");
System.out.println(celsius);

BoilingPoint error = BoilingPoint.valueOf("no");

BoilingPoint[] values = BoilingPoint.values();
System.out.println(Arrays.toString(values));
```

Java enums have lots of features that other Java classes don't, including the ability to fetch an array of the enumerated values.

- Open the `StarWarsBattle` class and navigate to the `main` method.
  - Convert each value of the `DamageType enum` to a string and print each to standard output.
  - Use the `valueOf(String)` method to convert a string into a value of each of the enumerated values and print each to standard output.
  - Finally, use the `values()` method to get an array of `DamageType` values and print it to standard output. ***Hint:*** use the `Arrays.toString(array)` method to convert the array into a nicely formatted string.
  - ***Challenge***: override the `toString()` method to return a string in the format `"<type> (<color>)"` e.g. `"HEAVY (Green)"`.

The `static` Modifier

- In the last unit, we used the `static` modifier before all of the functions that we wrote, but what does `static` mean?
- The `static` modifier indicates that only ***one global*** instance of the corresponding item will exist in the program.
  - It may be used to modify ***methods*** or ***fields***.
- The side effects of this are:
  - The static item is part of the ***class*** and not an instance.
  - You can ***access*** it using `class.staticMember` instead of `object.member`
    - It is considered bad practice to access a static member using an object.
  - If a static member is changed, it is changed ***globally***.
  - Static methods can only access other ***static*** methods and fields.
- In general, you should avoid using `static` outside of `main` and when creating constants.

```
1  public static void main(String [] args) {
2      Pet myPet = new Pet("Cosmo", 2,
3                          Species.DOG);
4
5      myPet.adopt("Bandit");
6      myPet.birthday();
7      System.out.println(myPet);
8  }
```

A method that uses the `static` modifier is in a ***static context*** and can only directly access other methods or fields that are also static.

Non-static fields and methods will need to be accessed through an instance of the class.

You should ***not*** make a non-static method `static` just so that it can be called from a static method.

`final` & Constants

Constants are both static and final and may also be public if they are needed outside of the class.

```
1  public class Circle {
2      public static final double PI = 3.14;
3
4      private final double radius;
5
6      public Circle(double radius) {
7          this.radius = radius;
8      }
9
10     public double circumference() {
11         return PI * radius * 2;
12     }
13
14     public double area() {
15         return PI * radius * radius;
16     }
17 }
```

Constants make better ***self documenting code*** and it is much easier to modify if the value of the constant has to change for any reason.

- The `final` modifier indicates that, once a value has been assigned to a variable, the value may never be changed.
  - A field that is final ***must*** be assigned a value when the object is constructed.
  - If a value is not assigned in the constructor or when the field is declared, the class will not compile.
- A ***constant*** is a variable that has a predefined value that never changes.
  - e.g. Pi (3.14159), the number of feet in a mile (5,280), etc.
- A constant is ***both*** `static` and `final`.
- In general, constants should be used instead of "magic numbers."
  - A "***magic***" number is one that just appears in code without any context.
  - This is especially true if the value is used in more than one place.
- Using constants makes it ***much*** easier to change the value.
  - For example, if we need to add 3 more digits of precision to Pi, we only need to change it in one place.

2.14 Look at all the Colors

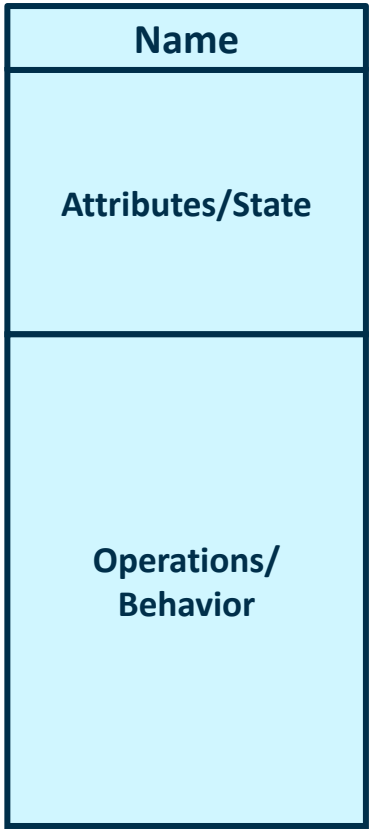
The current version of the `StarWarsBattle` uses hardcoded string literals for the names of the ships and weapons. Let's practice using constants by declaring a constant for each ship and weapon name and reusing it when the corresponding ship is created.

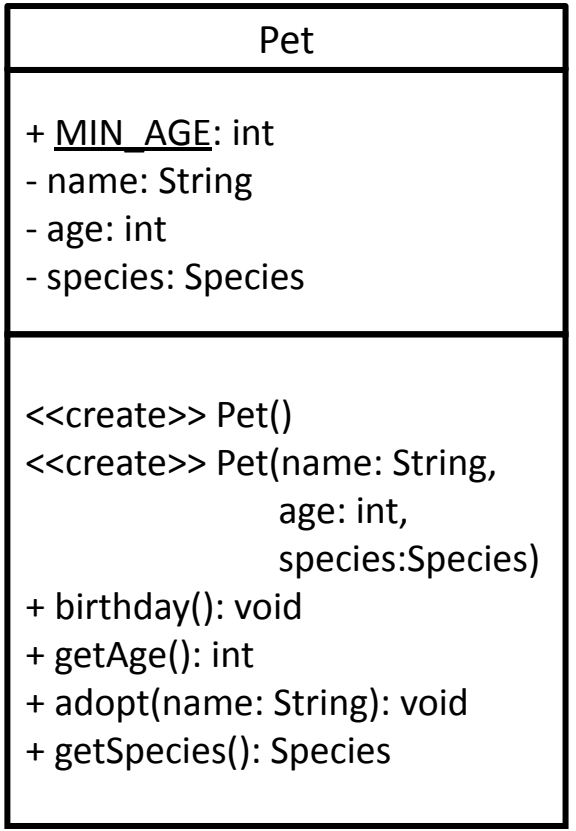
```java
public class Circle {
    public static final double PI = 3.14;

    public double circumference() {
        return PI * radius * 2;
    }

    public double area() {
        return PI * radius * radius;
    }
}
```

- In `StarWarsBattle`, add constants for each of the ship and weapon names.
    - It is recommended the constant names be based off the value, e.g. `X_WING` and not `SHIP_1`.
- Change any code that was using the string literals to use the new constants.
- In the main method inside `StarWarsBattle`, create your ships and print the details about the ship and weapon names to make sure that everything looks correct.

UML Class Diagrams

- The ***Unified Modeling Language*** (***UML***) is the standard used to ***diagram*** object-oriented software systems.
- While UML can be used to diagram many different aspects of a software system, the most common diagram is a ***class diagram***.
- Boxes partitioned into three parts are used to describe each class including:
  - The class ***name***, its ***attributes*** (fields), and its ***operations*** (methods).
- Visibility is specified for each attribute and operation:
  - **+** public
  - **-** private
  - **#** protected





A Closer Look at UML Class Diagrams

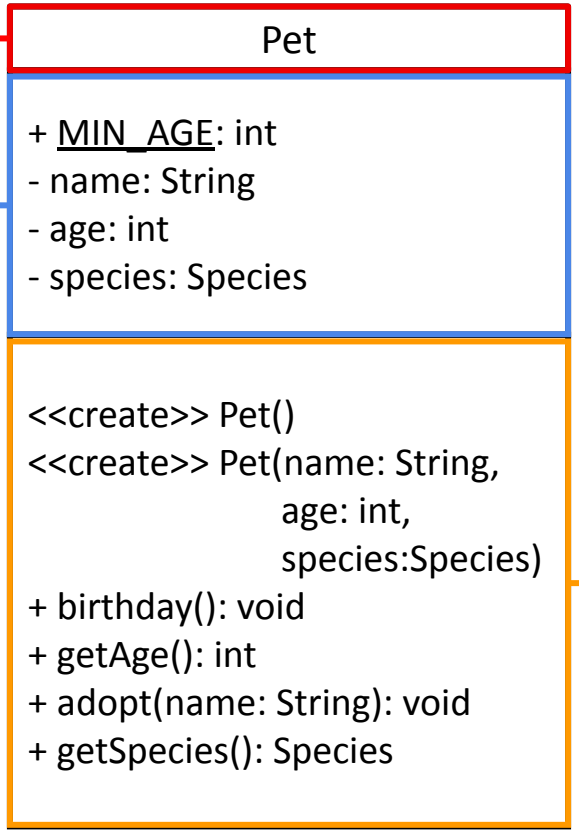
Classes in a UML class diagram are represented by boxes divided into three parts.

The class ***name*** is written in the topmost partition.

The ***attributes*** (fields) are listed in the middle partition.

Attributes are shown in with the name first followed by the type, e.g.
`age: int`

Static attributes are *underlined* and by convention are named in UPPERCASE.



The ***operations*** (methods) re listed in the bottom partition.

Any parameters are shown with the name first followed by the type, e.g.
`name: String`

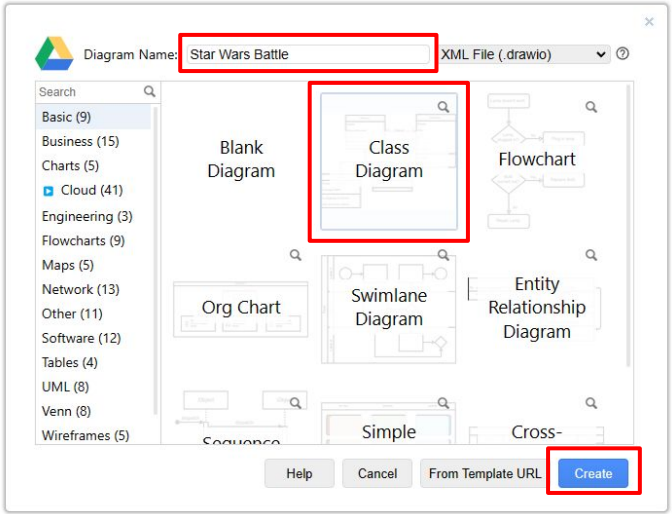
The return value is listed after the method declaration, e.g.
`getAge(): int`

Constructors are marked with the `<<create>>` annotation.

Visibility is indicated using **+** (public), **–** (private), or **#** (protected).

2.15 A UML Drawing Tool

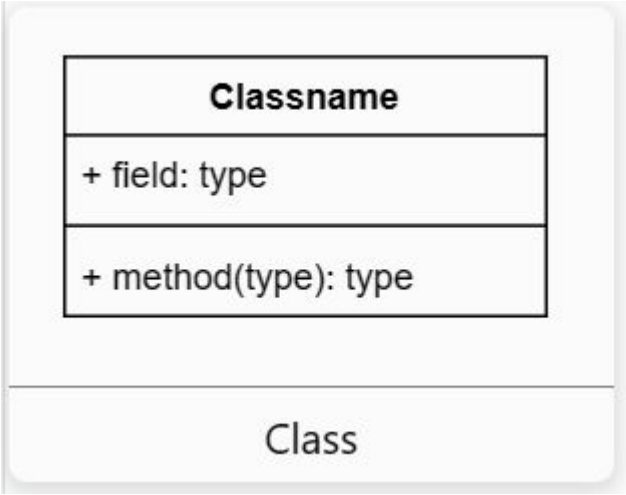
While many UML class diagrams are drawn by hand (e.g. on paper or a whiteboard), professional software engineers document their designs using software tools. There are many tools available, but today we will be using a free, online tool that connects to Google Drive.



- Open a browser and navigate to https://draw.io
  - You will be asked where your diagrams should be saved. Click the ***Google Drive*** option.
  - You will be asked to ***authorize*** draw.io so that it can save files to your Google Drive.
  - Log in with your RIT Gmail and password. Don't forget the "***@g***" in your email, e.g. "abc1234***@g***.rit.edu"!
- Now you are ready to create your first diagram!
  - Choose the ***Create New Diagram*** option.
  - In the next dialog, type "Star Wars Battle" into the ***Diagram Name*** field at the top and then choose the ***Class Diagram*** option.
  - Next, click the ***Create*** button and choose your root folder.
- You are now ready to create your first UML class diagram!

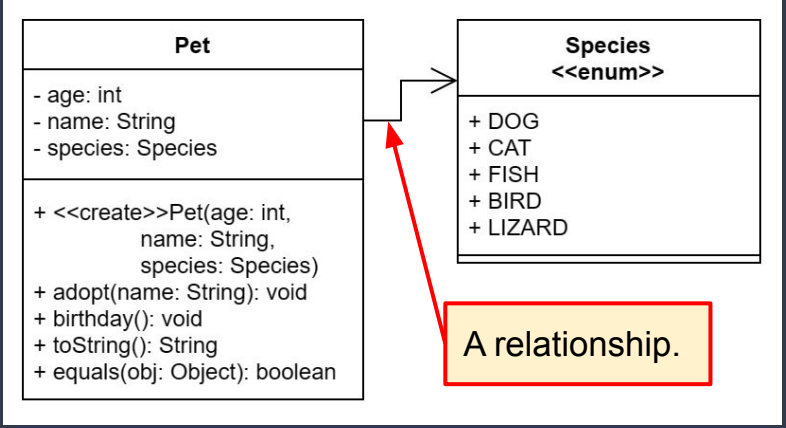
2.16 A First UML Class Diagram

Being able to read and draw UML is an essential skill for any software developer. Professionals must document their software design and keep it up to date so that they can share it with their teams, customers, and other stakeholders. Let's practice now!



- Start by deleting any classes that were created for you in the new diagram.
- Draw a UML class diagram representing the `Weapon` class.
  - Expand the ***UML*** palette on the left. It may be all the way at the bottom.
  - Click and drag the "***Class***" shape into the drawing area. The shapes may be very hard to read, but when you mouse over each shape you will see a larger pop-up.
  - Edit the new shape to change the name, list the fields, and list the operations. Don't forget to indicate visibility where appropriate!
  - Refer to the "Closer Look" slide while drawing!
- Once you are satisfied with your diagram, ask your instructor or a Course Assistant to check it over.
- Next, Draw a UML class diagram representing the `Ship` class.

UML – Relationships

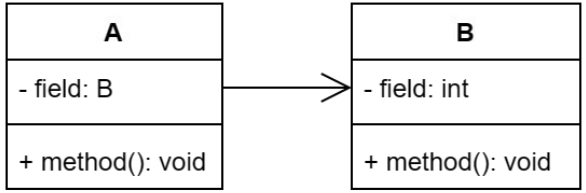


UML includes several different kinds of associations, each of which has a different meaning.

Professional software developers are expected to be able to implement code based on UML class diagrams.

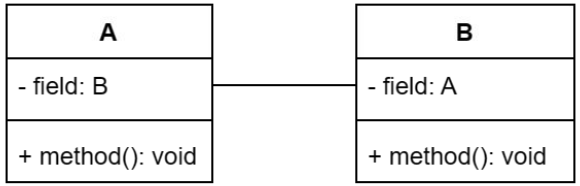
- In addition to describing individual classes, UML is used to describe the relationships that exist between classes.
  - A relationship between two classes indicates that the classes use each other in some way.
- Relationships are represented using a line that connects the two classes.
  - The lines may be solid or dashed.
  - A solid line indicates a stronger relationship.
- Relationships may also include an arrow that indicates ***directionality***.
  - The arrow points ***from*** a class ***to*** the class that it uses.
  - If there is no arrow, the relationship is ***undirected*** meaning that the classes ***use each other***.
- Class diagrams and relationships form a more complete picture of how all the components relate to each other.
  - The association syntax is very important, using the wrong ones is like using the wrong grammar or punctuation in a sentence; it totally changes the meaning, e.g. "Let's eat Grandma!" vs "Let's eat, Grandma."

Basic UML Relationships

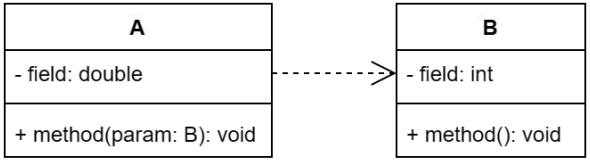


An ***association*** usually indicates that class ***A*** has a ***field*** of type ***B***.

Note the ***solid line*** and that the arrow points ***from*** A ***to*** B because A needs B (not the other way around).



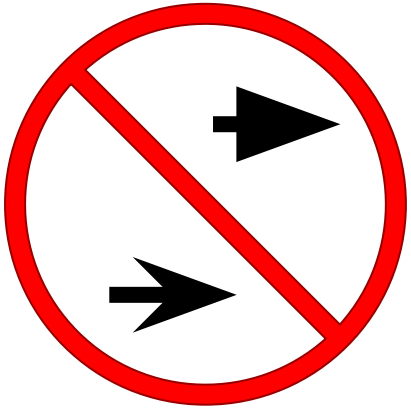
The absence of an arrow indicates that the association is ***undirected*** - both classes use ***each other***.



A ***dependency*** is weaker than an association and indicates that ***A uses B*** in some way, e.g. as a ***parameter*** or a ***return value*** but B is ***not*** part of the state of A.

Note the ***dashed line*** and that the arrow points in the direction of dependency: A depends on B (not the other way around).

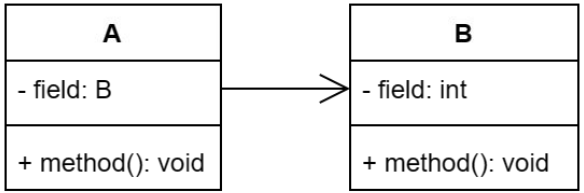
A dependency may also be undirected if both classes use each other. In that case there is no arrow on the dashed line.



These kinds of arrows are ***never*** used in UML class diagrams.

2.17 Adding Relationships

It is usually not enough to represent individual classes in a UML class diagram. The relationships (associations, dependencies, etc.) between classes must also be represented. Let's practice by updating your UML class diagram to include the correct relationships between classes.

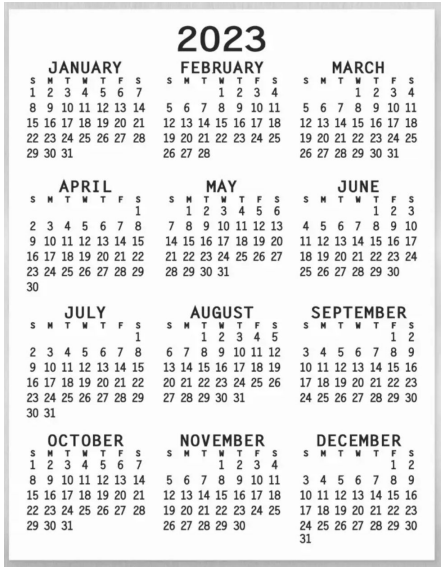


There are many different kinds of relationships in UML. We will explore more later this semester.

- Connect the `Ship` and `Weapon` classes using the correct kind of UML relationship.
  - Refer to the Basic UML Relationships and decide which you think is most appropriate.
  - Should it be directed or undirected?
  - Once you have decided, use the UML palette to drag the correct kind of arrow into your diagram.
  - Click the arrow to select it and drag the anchors to connect your two classes together.
- Next, add classes to represent the `DamageType` `enum` and `StarWarsBattle`.
  - Connect them to the other classes in your diagram using the correct UML relationships.

2.18 Days in a Year

Let's practice some of the basics! The number of days in a year is usually 365, but a leap year has one extra day. Write class that provides a method that, given a year number, returns the number of days in that year.

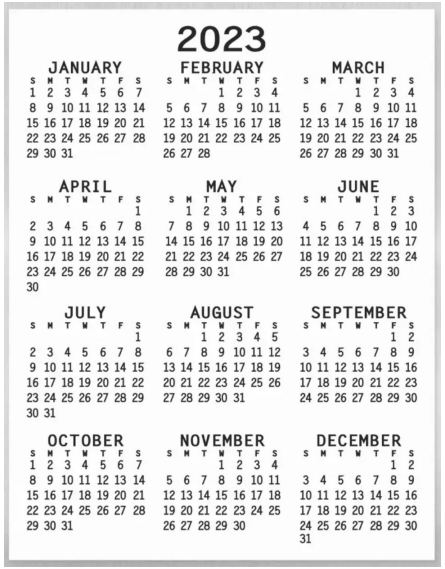


If needed, you may refer to the code that you wrote for earlier activities in this unit.

- Create a new Java class in a file named "`Year.java`".
  - ***Add*** a `static` ***field*** named `DAYS_IN_YEAR` to hold the number of days in a typical year, i.e. `365`.
  - ***Define*** a `static` method called "`daysInYear`" that, given an `int` year, will return the correct number of days in that year. A ***leap year*** is any year that:
    - ***Is*** divisible by 400.
    - ***Is*** divisible by 4 but ***not*** divisible by 100.
- Define a `main` method with the appropriate signature.
  - ***Call*** your `daysInYear` method with several different years and print the results, e.g. `"Days in 2021: 365"`.
- ***Run*** your class to make sure that it works correctly.
- What happens if you ***change*** `DAYS_IN_YEAR` from your `main` method ***before*** calling the `daysInYear` method?

2.19 Representing a Single Year

The Year class currently has a single static method. Let's improve it so that instances of the class can be created to represent an individual year.

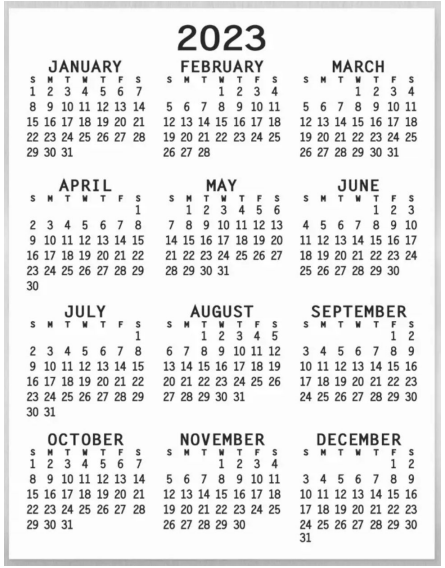


If needed, you may refer to the code that you wrote for earlier activities in this unit.

- Open the `Year` class and make the following changes:
  - ***Add*** a field to hold the value for the `yearNumber`.
  - ***Add*** a ***constructor*** that creates a new `Year` with a specified value.
  - ***Add*** an ***accessor*** for the `yearNumber`.
  - Does it make sense to add a mutator as well?
  - ***Define*** a method named "`numberOfDays`" that returns the number of days for this year.
    - ***Hint:*** use the `static` `daysInYear` method to compute the value.
  - Make sure to use ***proper encapsulation***!
- In `main`, create several instances of your class to represent different years.
  - Print each to standard output.

2.20 Immutable Fields

Once a value is assigned to a variable that uses the `final` modifier, that value can never change. At least one of the fields inside the `Year` class should be `final`. Update the class to make it so.

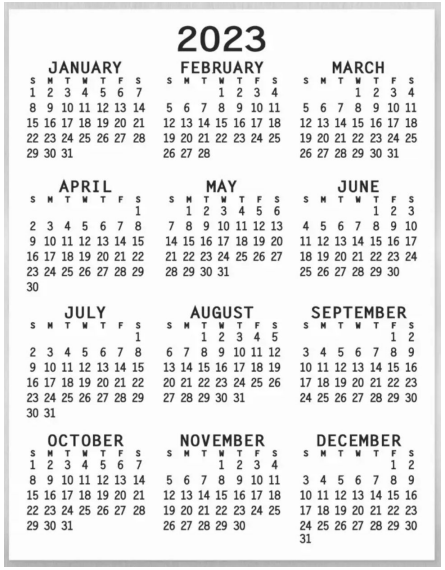


- Open the `Year` class and navigate to the `DAYS_IN_YEAR` variable.
  - Use the `final` ***modifier*** to make it a constant (immutable).
  - This may cause compilation problems elsewhere in your code. Comment out any broken code.
- Remember, `final` is meant for values that, once they are set, should never change. Are there any other values in Year that should be modified with `final`?
- ***Run*** your class to make sure that everything is still working.

If needed, you may refer to the code that you wrote for earlier activities in this unit.

2.21 Adding Special Methods

Java includes a number of special methods, but the default implementations are not very useful. Let's continue to enhance the Year class by adding more useful implementations of some of those special methods.



- Add a special method that returns a string representation of the year in the format:
  - `"Year{yearNumber=<yearNumber>, days=<number of days>}"`
  - Test your new method in `main` by creating at least 3 different years and printing them to standard output.
- Add a special method to the Year class so that two individual years are considered equal if they have the same `yearNumber`.
  - Test the new method from `main` by creating at least one more year that is equal to one of the years that you previously created.
  - Print the results of comparing the new year to at least two other years.

If needed, you may refer to the code that you wrote for earlier activities in this unit.

JavaDoc

An example of a ***JavaDoc comment*** used to document a method including `@parameter`, `@return`, and `@exception` tags.

```java
/**
 * Computes and returns the factorial of n.
 *
 * @param n The integer input.
 *
 * @return The factorial of n.
 *
 * @exception IllegalArgumentException If n < 0.
 */
public static int factorial(int n) {
    if(n < 0) {
        throw new IllegalArgumentException(
            "n cannot be negative!");
    } else if(n == 1 || n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Refer to the style example for more JavaDoc examples.

- ***JavaDoc*** is a special kind of comment that can be used to document ***classes***, ***methods***, and ***fields***.
  - A JavaDoc is a ***multi-line comment*** that begins with `/**` and ends with `*/`.
  - When you hit enter after typing `/**`, VS Code will helpfully ***stub out*** the rest of the JavaDoc comment.
- A JavaDoc comment should ***immediately precede*** whatever it is being used to document.
- JavaDoc also supports many special ***tags*** - special reserved words starting with an `@`.
  - For example, an `@author` tag can be included with the JavaDoc for a class to identify the author of the class.
- There are some special tags that can be used with JavaDoc documenting a method.
  - `@param <name>` documents the purpose parameter with the specified name.
  - `@return` documents the return value (if there is one).
  - `@exception` documents an exception that may be thrown by the method.
- You should ***always*** use JavaDoc to document your classes and any ***public*** fields, methods, and constructors.

2.22 JavaDoc of the Year

Documentation is an important part of software development. You will be expected to write JavaDoc for all classes, and any public methods or fields. Start practicing now by adding JavaDoc to the `Year` class.

```java
/**
 * Computes and returns the factorial of n.
 *
 * @param n The integer input.
 *
 * @return The factorial of n.
 *
 * @exception IllegalArgumentException If n < 0.
 */
public static int factorial(int n) {
    if(n < 0) {
        throw new IllegalArgumentException(
            "n cannot be negative!");
    } else if(n == 1 || n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

- Open the `Year` class and add ***JavaDoc*** for:
  - The ***class***.
  - All ***fields***.
  - All ***methods***.
- This is practice! You should write JavaDoc even for private members.
- ***Hint***: if you type `/**` and hit ***enter***, VS Code will helpfully stub out the JavaDoc for you, including `@param` and `@return` tags for you to fill in.

String Parsing

- You should recall that Python string type provides some methods that can be used for basic ***string parsing***.
- At this point it should probably come as no surprise that Java's String class also provides similar methods.
  - `String t = s.trim()` returns a ***copy*** of the string with any leading and trailing ***whitespace removed*** (including newlines). Whitespace is ***not*** removed from the middle of the string. It is similar to Python's `strip()` method.
  - `String[] tokens = s.split(regex)` will use the specified ***regular expression*** to split a string into an array of tokens. It is similar to Python's `split()` method.

The `String` class provides ***many*** methods that can be used for ***string parsing***. One such method is `trim()`, which will return a copy of the string with ***whitespace removed*** from both ends (but ***not*** the middle).

```
1  String parseMe = "\t  The.Quick.Brown.Fox\n";
2  String trimmed = parseMe.trim();
3  String[] tokens = trimmed.split("[.]");
4  System.out.println(Arrays.toString(tokens));
```

Another example is the `split(String regex)` method, which uses a ***regular expression*** to split the string into an ***array of tokens***.

Some regular common regular expressions include `""` (which will split into ***individual characters***) and `" "` (which will split on ***spaces***).

2.23 Parsing Years

String parsing refers to extracting data from a string. Often this means converting the string data into some other form. Let's practice string parsing in Java by parsing strings into `Year` objects.

```
String parseMe = "\t  The.Quick.Brown.Fox\n";
String trimmed = parseMe.trim();
String[] tokens = trimmed.split("[.]");
System.out.println(
   Arrays.toString(tokens));
```

- Open the `Year` class and add a static method named "`parseYears`" that declares a parameter for a string of `years` and ***returns an array*** of `Year` objects.
  - Assume the string is in the format "`<year 1> <year 2> … <year n>`" where `n` is the number of years in the string, e.g. "`1975 2000 1952 2023`"
  - Use `Integer.parseInt(String)` to convert the tokens into integers.
  - ***Do not*** worry about ***error handling***; assume the string is correctly formatted.
- Test your new method by calling it from `main` with the properly formatted string of your choice.
  - Use `Arrays.toString()` to print the array that is returned.

- It is often the case that the programs that we write will need some elements of ***randomness***.
  - We accomplished this in Python by importing the `random` module and calling the `randint(a, b)` and `randrange(a, b)` functions.
- Thankfully, Java also provides a class that can be used to generate ***pseudorandom*** numbers: `java.util.Random`
- An instance of the `Random` class can be created with or without a ***seed***, i.e.
  - `Random RNG = new Random();`
  - `Random RNG = new Random(1);`
- Once created, an instance of the `Random` class can be used to generate pseudorandom integers between an ***origin*** and ***some upper bound - 1***, e.g. `RNG.nextInt(50, 100)` will generate a number between ***50*** and ***99***.
- Random also includes a method `nextInt(bound)` that returns an integer between ***0*** and ***bound - 1***.

java.util.Random

The `Random` class is in the `java.util` package, and so should be ***imported***. An instance of the class can be created with or without a ***seed***.

```java
import java.util.Random;

public class RandomExample {
    private static final Random RNG =
        new Random();

    public static void main(String[] args) {
        // generate a number with a value
        // between 50 and 99
        int rand1 = RNG.nextInt(50, 100);

        // generate a number with a value
        // between 0 and 99
        int rand2 = RNG.nextInt(100);
    }
}
```

There are two different methods to use when generating pseudorandom numbers. In both cases the upper bound is ***exclusive***.

2.24 Random Years

Randomness is an essential part of many software programs. The `Random` class in Java can be used to generate pseudorandom numbers to simulate things like shuffling cards or rolling dice. Let's practice using it to generate random years.

```
import java.util.Random;

public class RandomExample {
    private static final Random RNG =
       new Random();

    public static void main(String[] args) {
        // generate a number with a value
        // between 50 and 99
        int rand1 = RNG.nextInt(50, 100);

        // generate a number with a value
        // between 0 and 99
        int rand2 = RNG.nextInt(100);
    }
}
```

- Open the `Year` class and define a new static method named "`getRandomYear`" that returns a new `Year` object representing a random year between `1900` and `2023`.
  - You will need to ***import*** the `java.util.Random` class.
  - Remember, the `nextInt(int origin, int bound)` method returns an integer ***between*** `origin` ***and*** `bound-1`. You will need to compensate for this to generate a number between `1900` and `2024`.
  - ***Challenge***: use constants for the minimum and maximum years.
- ***Update*** your `main` method to generate a few random years and print them to standard output.

java.lang.Object

Remember, when declaring a variable you must specify a ***type***. From that point forward, you can only assign values that are ***compatible*** with that type to the variable.

Because ***every*** reference type is related to the `java.lang.Object` class, instances of ***every*** reference type are compatible with `Object`!

This means that instances of ***any*** class can be ***assigned to*** or used as ***arguments*** and ***return values*** where `Object` is declared as the type.

```
1  public Object objectCode(Object obj) {
2      String s = "\t\tabcdefg" + obj;
3      Object o = s + s;
4      String x = (String)o;
5      return x.trim();
6  }
```

- As we have discussed, there are ***two*** basic kinds of types in Java.
  - ***Primitive types*** like `byte`, `int`, `boolean`, and `float`.
  - ***Reference types***, which comprise ***every*** type that is ***not*** a primitive type.
- The `java.lang.Object` class is a very special kind of reference type: ***every*** reference type in Java is ***related*** to `Object`.
  - It also means that an instance of ***any*** type can be used in place of the `Object` class in code.
  - This is why, for example, that an instance of ***any class*** can be used as the argument to the `equals(Object)` method.
  - We'll explain exactly why this is the case in the ***inheritance unit***.
- The `Object` class also provides the ***default implementations*** of the `toString()` and `equals(Object)` methods.
  - If a class does not explicitly ***override*** these methods, then `Object`'s implementations are used.
- It also provides many other useful methods that we will discuss later in the semester.
  - Some of these methods are `final`, which means that the ***cannot*** be overridden by other classes.

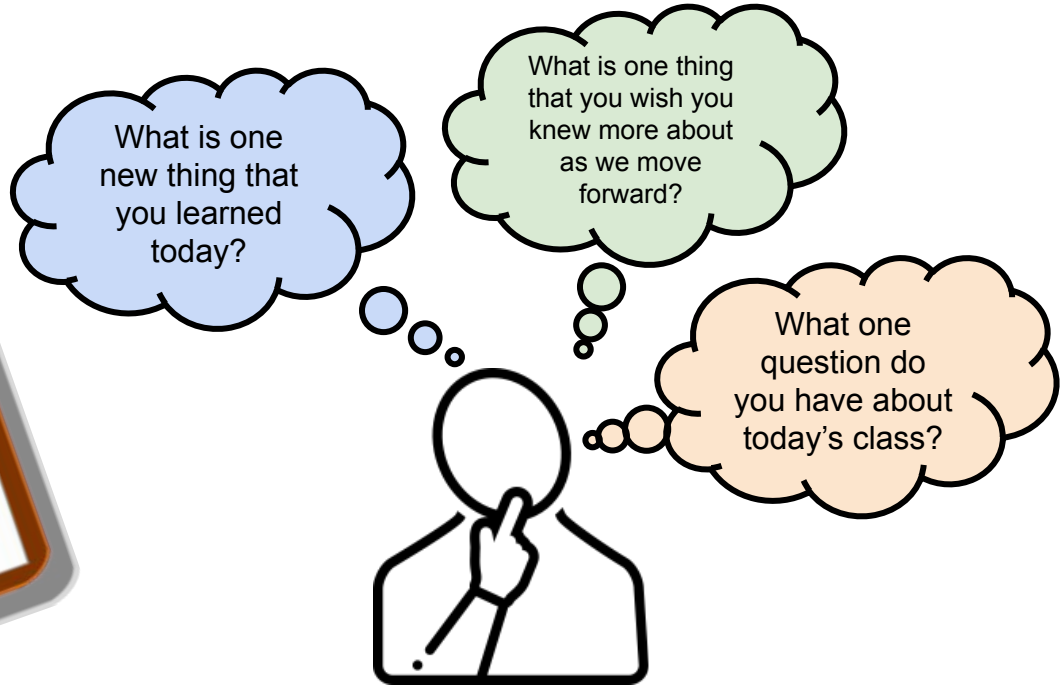
2.25 Objects in Space

In Java, every non-primitive type is an Object. Values of any type may be assigned to an Object variable, passed as arguments to an Object parameter, or store in an array of Objects. Experiment with the Object class to see how this works in practice.

```
public Object objectCode(Object obj) {
    String s = "\t\tabcdefg" + obj;
    Object o = s + s;
    String x = (String)o;
    return x.trim();
}
```

- Create a new class named `Objects` and define a static method named "`funWithObjects`" that declares an `Object` parameter and returns an `Object` array.
  - Create an Object array large enough to hold at least 5 values.
  - Convert the Object parameter into a `String` and store it in the first index in the array. ***Hint***: call its `toString()` method!
  - Fill the remaining indexes in the array with values of different types.
  - What happens when you try to store a primitive value in the array?
  - What about an instance of one of your own classes, e.g. `Year`?
  - What about another array?
- Define a `main` method with the appropriate signature.
  - Call the `funWithObjects` method at least 3 times with arguments of different types.
  - Use the `Arrays.toString(array)` method to print the return values to standard output.

Summary & Reflection



Please answer the questions above in your notes for today.