Learning Objectives - Classes and Objects

- **Define the terms class, objects, instance, and instantiate**
- **Identify the difference between classes and objects**
- **Create a user-defined object**
- **Add an attribute to an object with dot notation**
- **Define class attribute**
- **Explain the difference between shallow and deep copies**

Built-In Objects

The String Object

You have already been using built-in Java objects. Strings are an example of a Java object.

```
    //add code below this line

    String s = new String("I am a string");
    System.out.println(s.getClass());

    //add code above this line
```

challenge

Try these variations:

Explore some of the methods associated with the string class.
* Add the line of code `System.out.println(s.isEmpty());`
* Add the line of code `System.out.println(s.getBytes());`
* Add the line of code `System.out.println(s.endsWith("g"));`

Java says that the class of `s` is `java.lang.String` (which is a string). Add the following code and run the program again.

```
    //add code below this line

    String s = new String("I am a string");
    System.out.println(s.getClass());

    Method[] methods = s.getClass().getDeclaredMethods();
    for (Method m : methods) {
      System.out.println("Method name: " + m + "\n");
    }

    //add code above this line
```

The variable `methods` is an array of all methods associated with the `String` class. If you look carefully at the output, you may be confused by the information on the screen. However, a few things, like `toUpperCase` and `toLowerCase`, may seem familiar. Methods will be covered in a later lesson, but it is important to understand that a string is not a simple collection of characters. Because a string is a class, it is a powerful way of collecting and modifying data.

Vocabulary

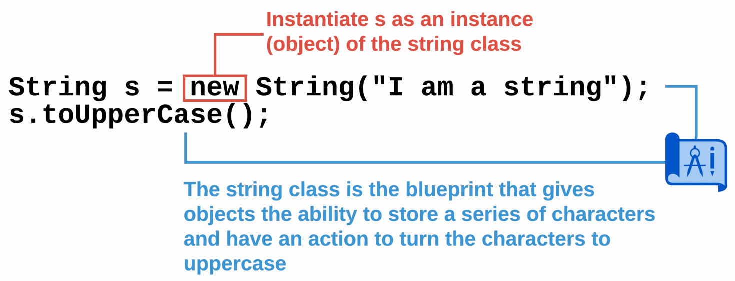
In the text above, the words “class” and “object” are used in an almost interchangeable manner. There are many similarities between classes and objects, but there is also an important difference. Working with objects has a lot of specialized vocabulary.

Classes - Classes are a collection of data and the actions that can modify the data. Programming is a very abstract task. Classes were created to give users a mental model of how to think about data in a more concrete way. Classes act as the blueprint. They tell Java what data is collected and how it can be modified.

Objects - Objects are constructed according to the blueprint that is the class. In the code above, the variable `s` is a string object. It is not the class. The string class tells Java that `s` has methods like `length`, `concat`, and `replace`. When a programmer wants to use a class, they create an object.

Instance - Another way that programmers talk about objects is to say that an object is an instance of a particular class. For example, `s` is an instance of the `String` class.

Instantiation - Instantiation is the process where an object is created according to blueprint of the class. The phrase “define a variable” means to create a variable. The variable is given a name and a value. Once it has been defined, you can use the variable. With objects, you use the phrase “instantiate an object”. That means to create an object, give it a name, store any data, and define the actions the object can perform.



```
String s = new String("I am a string");
s.toUpperCase();
```

Instantiate s as an instance (object) of the string class
The string class is the blueprint that gives objects the ability to store a series of characters and have an action to turn the characters to uppercase

Class vs Object

User-Defined Objects

Defining an Object

Assume you want to collect information about actors. Creating a class is a good way to keep this data organized. The `class` keyword are used to define a class. For now, do not add anything as the body of the class.

```java
//add class definitions below this line

class Actor {

}

//add class definitions above this line
```

▼ **Naming classes**

The convention for naming classes in Java is to use a capital letter. A lowercase letter will not cause an error message, but it is not considered to be “correct”. If a class has a name with multiple words, all of the words are pushed together, and a capital letter is used for the first letter of each word. This is called camel case.

Classes are just the blueprint. To you use a class, you need to instantiate an object. Here is an object to represent Helen Mirren. Be sure to put this code in the `main` method.

```java
    //add code below this line

    Actor helen = new Actor();
    System.out.println(helen.getClass());

    //add code above this line
```

So you now have `helen`, which is an instance of the `Actor` class.

Adding Attributes

The point of having a class is to collect information and define actions that can modify the data. The `Actor` class should contain things like the name of the actor, notable films, awards they have won, etc. These pieces of information related to a class are called attributes. Attributes are declared in the class itself. The example below adds the `firstName` and `lastName` attributes which are both strings.

```
//add class definitions below this line

class Actor {
  String firstName;
  String lastName;
}

//add class definitions above this line
```

You can change the value of an attribute with the assignment operator, `objectName.attribute = attributeValue`. Notice that you always use `objectName.attribute` to reference an attribute. This is called dot notation. Once an attribute has a value, you can treat it like any other variable. Add the following code to the `main` method. You are assigning values to the attributes `fistName` and `lastName`, and then printing these values.

```
    //add code below this line

    Actor helen = new Actor();
    helen.firstName = "Helen";
    helen.lastName = "Mirren";
    System.out.println(helen.firstName + " " + helen.lastName);

    //add code above this line
```

challenge

Try these variations:

- Change the print statement to:

```
System.out.println(helen.firstName.toUpperCase() + " " +
        helen.lastName.toLowerCase());
```

- Add the attribute `totalFilms` and assign it the value `80`
- Add the print statement `System.out.println(helen);`

▼ **Code**

You may have noticed that printing the object `helen` returns `Actor@` followed a series of numbers and letters. Java is telling you that `helen` is an object of class `Actor` and the numbers and letters represent the object's location in memory.

```java
import java.lang.Class;

//add class definitions below this line

class Actor {
  String firstName;
  String lastName;
  int totalFilms;
}

//add class definitions above this line

public class UserDefined {
  public static void main(String[] args) {

    //add code below this line

    Actor helen = new Actor();
    helen.firstName = "Helen";
    helen.lastName = "Mirren";
    helen.totalFilms = 80;
    System.out.println(helen.firstName.toUpperCase() + " "
      + helen.lastName.toLowerCase());
    System.out.println(helen.totalFilms);
    System.out.println(helen);

    //add code above this line
  }
}
```

The Constructor

Too Much Code

Imagine that the `Actor` class has more attributes than on the previous page.

```java
//add class definitions below this line

class Actor {
  String firstName;
  String lastName;
  String birthday;
  int totalFilms;
  int oscarNominations;
  int oscarWins;
}

//add class definitions above this line
```

Now create a object for Helen Mirren with values for each attribute. Adding each attribute individually requires several lines of code. This is especially true if you more than one instance of the `Actor` class.

```java
    //add code below this line

    Actor helen = new Actor();
    helen.firstName = "Helen";
    helen.lastName = "Mirren";
    helen.birthday = "July 26";
    helen.totalFilms = 80;
    helen.oscarNominations = 4;
    helen.oscarWins = 1;
    System.out.println(helen.firstName + " " + helen.lastName);

    //add code above this line
```

The class `Actor` creates a class and its attributes. It does not assign value to any attributes; the user has to do this. A class is suppose to be a blueprint. It should lay out all of the attributes and their values for the user. Classes can do this when you use the constructor.

The Constructor

The constructor is a special method for a class. Its job is to assign value for attributes associated with the object. These attributes can also be called instance variables. In Java, the constructor is the class name, parentheses, and curly brackets. Inside of the constructor, give attributes their values.

```java
//add class definitions below this line

class Actor {
  String firstName;
  String lastName;
  String birthday;
  int totalFilms;
  int oscarNominations;
  int oscarWins;

  Actor() {
    firstName = "Helen";
    lastName = "Mirren";
    birthday = "July 26";
    totalFilms = 80;
    oscarNominations = 4;
    oscarWins = 1;
  }
}

//add class definitions above this line
```

Instantiating `helen` as an instance of the `Actor` class automatically calls the constructor. Since the instance variables (attributes) have values, you can remove those lines of code from the `main` method.

```java
    //add code below this line

    Actor helen = new Actor();
    System.out.println(helen.firstName + " " + helen.lastName);

    //add code above this line
```

challenge

Try these variations:

- Add this print statement to the `main` method:

```
System.out.println(helen.firstName + " " + helen.lastName +
        "\'s birthday is " + helen.birthday);
```

- Add this print statement to the `main` method:

```
System.out.println(helen.firstName + " " + helen.lastName +
        " won " + helen.oscarWins + " Oscar out of " +
        helen.oscarNominations + " nominations");
```

The Constructor and Parameters

The Constructor and Parameters

Now imagine that you want to use the `Actor` class to instantiate an object for Helen Mirren and Tom Hanks. Create the `Actor` class just as before.

```java
//add class definitions below this line

class Actor {
  String firstName;
  String lastName;
  String birthday;
  int totalFilms;
  int oscarNominations;
  int oscarWins;

  Actor() {
    firstName = "Helen";
    lastName = "Mirren";
    birthday = "July 26";
    totalFilms = 80;
    oscarNominations = 4;
    oscarWins = 1;
  }
}

//add class definitions above this line
```

Now instantiate two `Actor` objects, one for Helen Mirren and the other for Tom Hanks. Print the `fistName` and `lastName` attributes for each object.

```java
    //add code below this line

    Actor helen = new Actor();
    Actor tom = new Actor();
    System.out.println(helen.firstName + " " + helen.lastName);
    System.out.println(tom.firstName + " " + tom.lastName);

    //add code above this line
```

The constructor `Actor` class only creates an object with information about Helen Mirren. You can make the `Actor` class more flexible by passing it an argument for each of attributes in the constructor. Parameters for the constructor method work just as they do for user-defined methods, be sure to indicate the data type for each parameter.

```java
//add class definitions below this line

class Actor {
  String firstName;
  String lastName;
  String birthday;
  int totalFilms;
  int oscarNominations;
  int oscarWins;

  Actor(String fn, String ln, String bd, int tf, int on, int ow)
      {
    firstName = fn;
    lastName = ln;
    birthday = bd;
    totalFilms = tf;
    oscarNominations = on;
    oscarWins = ow;
  }
}

//add class definitions above this line
```

When you instantiate the two `Actor` objects, you can pass the constructor the relevant information for both Helen Mirren and Tom Hanks. The code should now print the correct first and last names.

```java
    //add code below this line

    Actor helen = new Actor("Helen", "Mirren", "July 26", 80, 4,
        1);
    Actor tom = new Actor("Tom", "Hanks", "July 9", 76, 5, 2);
    System.out.println(helen.firstName + " " + helen.lastName);
    System.out.println(tom.firstName + " " + tom.lastName);

    //add code above this line
```

challenge

Try these variations:

- Create an instance of the `Actor` class for Denzel Washington (December 28, 47 films, 8 nominations, 2 wins)
- Print the `birthday` and `totalFilms` attributes for the newly created object

▼ **Code**

Your code for the object representing Denzel Washington should look something like this:

```
        //add code below this line

        Actor denzel = new Actor("Denzel", "Washington",
          "December 28", 47, 8, 2);
        System.out.println(denzel.birthday);
        System.out.println(denzel.totalFilms);

        //add code above this line
```

Default Values

We can assume that the average actor probably has not been nominated or won an Oscar. So instead of making these attributes parameters for the constructor, we can give them the default value of 0. These attributes can always be updated later on.

```java
//add class definitions below this line

class Actor {
  String firstName;
  String lastName;
  String birthday;
  int totalFilms;
  int oscarNominations;
  int oscarWins;

  Actor(String fn, String ln, String bd, int tf) {
    firstName = fn;
    lastName = ln;
    birthday = bd;
    totalFilms = tf;
    oscarNominations = 0;
    oscarWins = 0;
  }
}

//add class definitions above this line
```

You can update the attributes once the object has been instantiated if need be.

```java
    //add code below this line

    Actor helen = new Actor("Helen", "Mirren", "July 26", 80);
    System.out.println(helen.oscarNominations);
    System.out.println(helen.oscarWins);

    helen.oscarNominations = 4;
    helen.oscarWins = 1;

    System.out.println(helen.oscarNominations);
    System.out.println(helen.oscarWins);

    //add code above this line
```

Class Attributes

Class Attributes

Up until now, the attributes created in the class are independent from one another when new objects are created. These are called object attributes. You can create an instance of the `Actor` class for Helen Mirren and another for Dwayne Johnson. Each object has different values for their attributes. A class attribute, however, is an attribute whose value is shared by each instance of a class.

To illustrate this, add the attribute `static String union` to the `Actor` class. The `static` keyword tells Java that this attribute is a class attribute. Set the value of `union` to the string `"Screen Actors Guid"`. Notice that class attributes are declared and initialized with a value outside of the constructor.

```java
//add class definitions below this line

class Actor {
  String firstName;
  String lastName;
  String birthday;
  int totalFilms;
  int oscarNominations;
  int oscarWins;
  static String union = "Screen Actors Guild";

  Actor(String fn, String ln, String bd, int tf) {
    firstName = fn;
    lastName = ln;
    birthday = bd;
    totalFilms = tf;
    oscarNominations = 0;
    oscarWins = 0;
  }
}

//add class definitions above this line
```

Create two instances of the `Actor` class and print the `union` attribute for both instances. Then change the attribute for one instances and print the `union` attribute for both instances.

```
    //add code below this line

    Actor helen = new Actor("Helen", "Mirren", "July 26", 80);
    Actor dwayne = new Actor("Dwayne", "Johnson", "May 2", 34);
    System.out.println(helen.union);
    System.out.println(dwayne.union);
    helen.union = "Teamsters";
    System.out.println(helen.union);
    System.out.println(dwayne.union);

    //add code above this line
```

Because this is a class attribute, the value is shared among all the instances. A change for instance, is a change for all instances.

challenge

Try this variation:

- Remove the keyword `static` and run the code again.

Class Attributes as Constants

Changing a class attribute can cause problems for other instances of the same class. That is why class attributes are often created as a constant. Remember, the Java convention for constants is to write the variable name in all caps.

```java
//add class definitions below this line

class Actor {
  String firstName;
  String lastName;
  String birthday;
  int totalFilms;
  int oscarNominations;
  int oscarWins;
  static final String UNION = "Screen Actors Guild";

  Actor(String fn, String ln, String bd, int tf) {
    firstName = fn;
    lastName = ln;
    birthday = bd;
    totalFilms = tf;
    oscarNominations = 0;
    oscarWins = 0;
  }
}

//add class definitions above this line
```

Java will now prevent users from changing the value of `UNION`. You should see an error message that Java cannot assign a value to a final variable.

```java
    //add code below this line

    Actor helen = new Actor("Helen", "Mirren", "July 26", 80);
    Actor dwayne = new Actor("Dwayne", "Johnson", "May 2", 34);
    helen.UNION = "Teamsters";

    //add code above this line
```

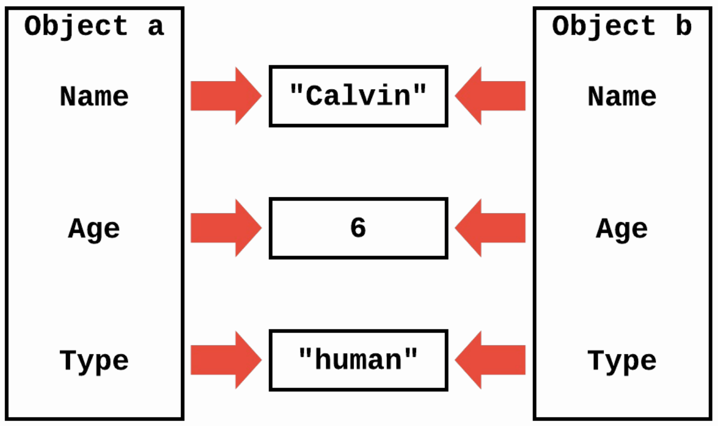
Shallow and Deep Copies

Shallow Copies

The code below will instantiate `a` as an instance of the `ComicBookCharacter` class. Object `a` will be given a name, an age, and a type. Object `b` will be a copy of `a`. Finally, the `name` attribute of object `a` is changed. What do you expect to see when the `name` attributes of objects `a` and `b` are printed?

```java
class ComicBookCharacter {
  String name;
  int age;
  String type;
}

//add class definitions above this line

public class Copies {
  public static void main(String[] args) {

    //add code below this line

    ComicBookCharacter a = new ComicBookCharacter();
    a.name = "Calvin";
    a.age = 6;
    a.type = "human";

    ComicBookCharacter b = a;
    a.name = "Hobbes";

    System.out.println("Object a name: " + a.name);
    System.out.println("Object b name: " + b.name);

    //add code above this line
  }
}
```

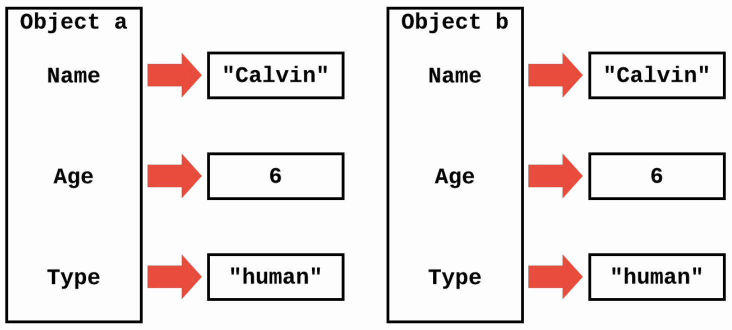
Both of the `name` attributes changed, even though the code only changed the `name` attribute of object `a`. This is because object `b` is a shallow copy of object `a`. Java makes a copy of object `a`, but object `b` shares the attributes with object `a`. That is why changing `name` for object `a` also affects the `name` attribute for object `b`.



Shallow Copy

Deep Copy

A deep copy is when Java makes a copy of object `a` and makes copies of each attribute. A deep copy keeps the attributes of one object independent of the other object. To create a deep copy, you are going to use something called a copy constructor.



Deep Copy

The copy constructor is a constructor that creates an object by initializing it with a previously created object of the same class. Start by creating another constructor that take a `ComicBookCharacter` as a parameter. Assign each object attribute with the corresponding attribute from the parameter.

```java
//add class definitions below this line

class ComicBookCharacter {
  String name;
  int age;
  String type;

  ComicBookCharacter(String n, int a, String t) {
    name = n;
    age = a;
    type = t;
  }

  ComicBookCharacter(ComicBookCharacter c) {
    name = c.name;
    age = c.age;
    type = c.type;
  }
}

//add class definitions above this line
```

Now instantiate `b` using the copy constructor. Object `b` will now be a deep copy of object `a`, and their attributes are independent from one another.

```java
    //add code below this line

    ComicBookCharacter a = new ComicBookCharacter("Calvin", 6,
        "human");

    ComicBookCharacter b = new ComicBookCharacter(a);
    a.name = "Hobbes";

    System.out.println("Object a name: " + a.name);
    System.out.println("Object b name: " + b.name);

    //add code above this line
```