

Developing the Model Layer

In previous lessons, we covered the software development process and started the first iteration through the development process. You defined the requirements by creating end user stories and then created a UML class diagram to document the design of classes needed in the model layer. We also set up of the development environment for your project.

We are now ready to continue the development process and enter into the construction phase by implementing the Model Layer classes defined in your UML class diagram in the last lesson.

You will need to know how to define variables and the different basic data types supported in Java before you can develop the classes in your UML class diagram. There are three categories of data types that can be used when defining variables in Java: primitives, classes, and lists.

In this lesson you will:

- Learn how to define variables in Java
- Learn the primitive data types in Java
- Learn about class data types
- Know what, why and how to develop Java Bean classes
- Learn about the different types of list in Java
- You will create your project
- Create packages to organize the classes in your project
- Use the IDE to develop the Java Bean classes defined in your UML class diagram
- Push and pull the project to and from the remote repository

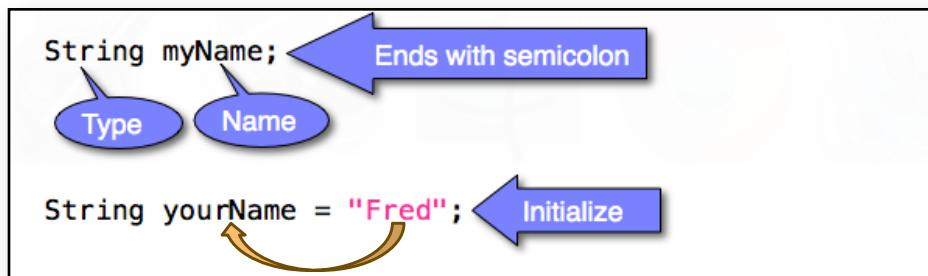
Defining variables in Java

There are specific rules and recommendations that need to be followed for naming and using variables in Java.

Java is a strongly typed language. This means that the first time you use a new variable name in Java you must define its data type. The data type of a variable may only be defined once. You specify the data type of the variable in front of the variable name followed by a semicolon as shown below.

```
dataType variableName;
```

Here are a few examples:

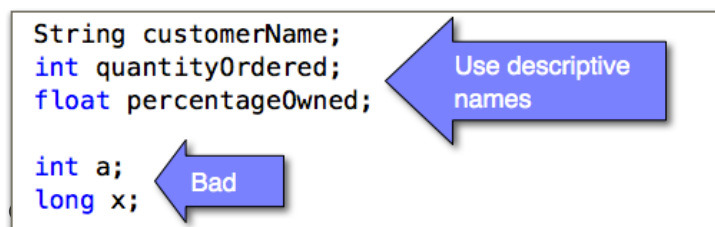


You optionally can assign an initial value to the variable as shown in the second example. The value to be assigned to the variable must always be on the right side of the equal sign. The equal sign in Java does not mean “equivalent to” as in mathematics. It means “**is assigned**” instead. The above statement can then read as follows from left to right:

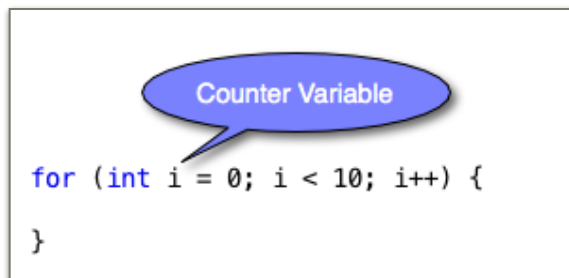
Of type `String`, variable `yourName` is declared and **is assigned** the value `"Fred"`

Naming variables

Giving meaningful, descriptive names to variables is important. The variable name should represent the data being stored in the variable. This will make it much easier for others to read your code and understand what your program is doing. In the example below, it is intuitive as to what the first three variables are representing.



Giving short non-descriptive names like `a` and `x` in the last two examples is considered a poor programming practice, because someone else reading your code will not have any idea what the variables are being used for. The exception to this is when defining a variable to act as a counter in a loop. It is traditional to just use a short variable name, such as `i` or `j` for a counter variable in a `for` loop.



Rules for naming identifiers in Java

Identifiers are the names used to identify a variable, method (function), constant or a class. Use these rules and guidelines when naming identifiers:

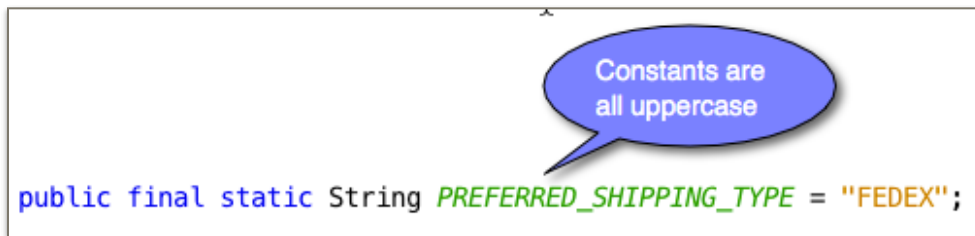
Rules

- Can only contain lower and upper case letters (a-z, A-Z), numbers (0-9), and the special characters, dollar sign and underscore. No other characters are allowed,
- Can not start with a number,
- Can not contain any blanks.

Guidelines

- Variable names should start with a lowercase letter except for variables defined as constants. Constant variables should be all uppercase.
- Most programmers use the Camel Case convention when the variable name is more than one word in length. Run all of the words together with no blanks, and start each word with an uppercase letter for each word except the first word. The first word should start with a lowercase letter.
- Class names should always start with an uppercase letter.

- Constant names should always be all uppercase with an underscore between names that contain multiple words.

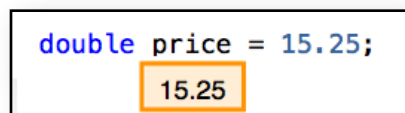


A diagram illustrating a Java constant declaration. The code `public final static String PREFERRED_SHIPPING_TYPE = "FEDEX";` is shown. A blue speech bubble points to the constant name `PREFERRED_SHIPPING_TYPE` with the text "Constants are all uppercase".

The Primitive Data Types

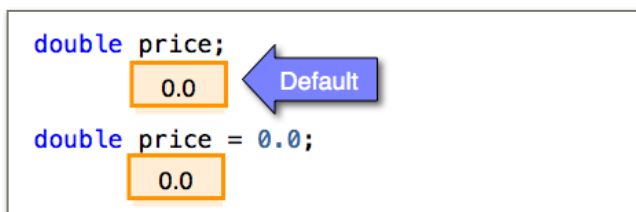
The most commonly used data types in Java are the primitive data types. They are used for variables when the value of the variable can be stored directly in the memory.

In the example below, the `price` variable is defined to be of the `double` data type and is assigned an initial value of `15.25`. Memory is assigned directly to the variable when it is first defined or used.



A diagram showing the declaration and assignment of a `double` variable. The code `double price = 15.25;` is shown. The value `15.25` is highlighted in an orange box.

A primitive variable may also be defined without assigning a value to it. The default value assigned to the variable is based on the data type of the primitive variable. In the example below, the first statement defines the variable `price` and assigns the default value of `0.0` to the variable.



A diagram showing two ways to declare a `double` variable. The first line is `double price;` with the value `0.0` highlighted in an orange box. A blue arrow labeled "Default" points from the text "Default" to the `0.0` value. The second line is `double price = 0.0;` with the value `0.0` highlighted in an orange box.

The second statement is equivalent to the first, but is more explicit and is thought to be a better programming practice.

The primitive data types in Java are divided up into four groups based on the type of data they represent: *Integer*, *Float*, *Boolean*, and *Character*.

The Integer Data Types

Integer data types are used to store whole numbers. They are stored in memory in a signed two's complement format. The default value of an integer type variable is always zero.

There are four integer data types: `byte`, `short`, `int` and `long`. The type you choose is based on the number of bits it uses in memory and the range of values that can be stored in a variable of that type. The greater the number of bits the greater the range of numbers that can be stored in that integer type.

`byte`

This type is often used for storing very large streams of small numbers or raw binary data that is to be sent across the network or written to disk.

Size	8 bits (1 byte)
Range of values	-128 (-2^7) to 127 (2^7-1)
Examples	<code>byte month;</code> <code>byte dayOfWeek = 5;</code>

`short`

May be useful when trying to save memory and the range of values is greater than what can be stored in a byte. This type is seldom used today.

Size	16 bits (2 bytes)
Range of values	-32,768 (-2^{15}) to 32,767 ($2^{15}-1$)
Examples	<code>short year;</code> <code>short dayOfYear = 225;</code>

`int`

This type is often used for integer data types, since the range of values is quite large and it only uses four bytes of memory.

Size	32 bits (4 bytes)
Range of values	-2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31}-1$)
Examples	<pre>int numberOfOrders; int numberOfCustomers = 110352968;</pre>

`long`

This is used for storing very large whole numbers. Many programmers use this data type for all integer numbers, because it helps prevent overflow errors where the calculated value goes beyond the range of values defined for the type. The extra four bytes of memory used is considered insignificant to the cost of fixing a bug as the result of an overflow error.

Size	64 bits (8 bytes)
Range of values	-9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 ($2^{63}-1$)
Examples	<pre>long population; int totalAutomobiles = 4321567933;</pre>

The Float Data Types

The float data type is used for storing decimal numbers, and for whole numbers that are beyond the range of values that can be stored in the `long` integer data type above. Java implements the IEEE-754 standard for using floating point numbers. Floating point numbers are represented using a fixed number of significant digits (called the mantissa) multiplied by the base 2 scaled to an exponent as shown in the example below.

$$1.334567 \times 2^{53}$$

Note: It is important to note that floating point numbers, like scientific notation, are actually a very close approximation of a value and may not always be accurate. Because of this, you should never use floating point numbers when dealing with currency or where exact accuracy is needed. Use the Java `BigDecimal` class for numbers that require absolute accuracy.

The default value for floating point numbers is zero and is expressed as `0.0f`. The character “f” placed at the end of a value in Java indicates that the constant number is a floating point number.

There are two types of floating point numbers, `float` and `double`. The type you choose is based on the number of bits in memory it takes to store the value, the amount of precision you need and the range of values that can be stored.

`float`

The `float` type is adequate for most applications using decimal numbers. The `float` type is referred to a *single-precision*. It is used for numbers that do not require as much precision and uses less memory.

Size	32 bits (4 bytes)
Precision	24 bits
Approximate range	-1.4e-045 to 3.4e+038
Examples	<code>float temperature;</code> <code>float weight = 145.34f;</code>

`double`

The `double` type is used when you needed greater precision or you need to store numbers beyond the range of values defined for the `float` type. The `double` type is said to have *double-precision*. Many programmers use this as the default type for floating point numbers, because it provides for more precision and avoids overflow errors.

Size	64 bits (8 bytes)
Precision	53 bits
Approximate range	-4.9e-324 to 1.8e+308
Examples	<pre>double weight; double rate = 0.134898672423d; double price = 4.99</pre>

The Boolean Data Type

The `boolean` type is used when you need a variable to store either store a `true` or `false` logical value.

Size	8 bits (1 bytes)
Possible values	<code>true</code> or <code>false</code>
Examples	<pre>boolean isValid; boolean inTheOffice = false;</pre>

The Character Data Type

The `char` type is used to store a single ASCII or Unicode character. Characters in Java are internally represented as a 16-bit number.

Size	16 bits (Unicode)
Possible values	<code>\u0000</code> (or 0) to <code>\uffff</code> (or 65,535 inclusive)
Examples	<pre>char grade; char direction = 'N';</pre>

Literal Values

Literal values are used when you want to assign a specific “hard coded” value to a variable. They are represented in a human-readable form. Here are some examples of literals used in Java:

```
15, 12.55, 20L, -7.15f, 42_83_4521, 'N', true, 0x75, 057, 0b1101
```

There is a unique way to express literal values for each of the different primitive data types and for strings.

Integers

The integer data types are expressed as follows. An `int` value is expressed simply as a whole number.

```
7, -5, 32755, 253
```

A `long` value is defined by adding either the letter ‘`l`’ or ‘`L`’ after the numbers.

```
-743L, 2101, 9246139128L
```

Optionally, you can place underscores in the number for readability. For example, to represent a phone number, or product number, etc.. The underscores are automatically removed by the compiler before being used.

```
208_496_8765, 489_98_1234L
```

Float

You express a floating point literal values by placing a decimal point in the number. It is assumed that the value is a `double` type unless you add the letter ‘`f`’ or ‘`F`’ after the number. In that case, the number is treated as the `float` data type.

```
0.59, .0125, 43.27f, 18.29F
```

You can add underscores for readability just as you can with integer literal values.

```
-1_235_943_21.78,148_376_250_893.4125f
```

boolean

There are only two possible literal values for a `boolean`.

```
true, false
```

char

Literal character values are expressed by enclosing a single letter with single quotes.

```
'D', 'O', 'G'
```

Strings

String literal values are expressed by enclosing a list of characters within double quotes.

```
"Have a good day", "Welcome to the party."
```

Escape sequences

There are special characters escape sequences that can be included within a string literal value that controls the display or printing of a string. The backslash character (`\`) is used at the beginning to indicate that the following character/s are part of an escape sequence. For example, the `\n` and `\t` escape sequences in the following string indicate the text is to be displayed on a new line and tabbed over.

```
"\n\tYou won the game"
```

Here is a list of the escape sequences that are supported in Java. They may be entered anywhere within a line.

Escape Sequence	Meaning
<code>\n</code>	Enter a new line
<code>\t</code>	Tab over horizontally
<code>\b</code>	Backspace

Escape Sequence	Meaning
\f	Form feed
\r	Carriage return
\'	Display literal single quote character
\"	Display literal double quote character
\\	Display literal backslash character
\###	Display the value in octal number format where the #'s represent actual octal digits between 0 - 7.
\uXXXX	Display the value in hexadecimal number format where the X's are actual hexadecimal digits between 0 - F.

Class Data Types

You create classes in Java to describe all of the other different types of persons, places or things that you want to model and use in your program. In a sense, each class describes a new unique type of data that contains a group of logically related variables and methods. Java comes shipped with a large number of predefined classes. In addition, you can create your own custom classes for your program. For example, you may create classes to describe a type of customer in a business or to describe the items that are for sale.

A Java class contains a definition of the variables and methods that are to be associated with a specific group of objects. Each object created from the class will have the same variables and methods. The variables defined in the class should be logically related to each other and describe the types of data to be stored in each object created from the class. A class may also contain methods that perform the actions that related to the object. It is important that all of the class variables and methods defined in a class support the single theme of the class just as the sentence in a paragraph should support the theme of the paragraph.

Here is an example of the structure of a typical class.

```
class ClassName {  
  
    // Definition of class variables  
    datatype variable1;  
    datatype variable2;  
    ...  
  
    // Constructor function  
    ClassName() {  
    }  
  
    // Other related functions  
    function1() {  
    }  
  
    function2() {  
    }  
    ...  
}
```

Each class must start with the `class` keyword followed by the name of the class. The class name should start with a capital letter. The class name must be followed by a block defined by the curly braces (`{}`). The class variables are typically defined at the top of the class followed by the constructor and the constructor must be the same name as the class. All of the other methods defined normally follow the constructor.

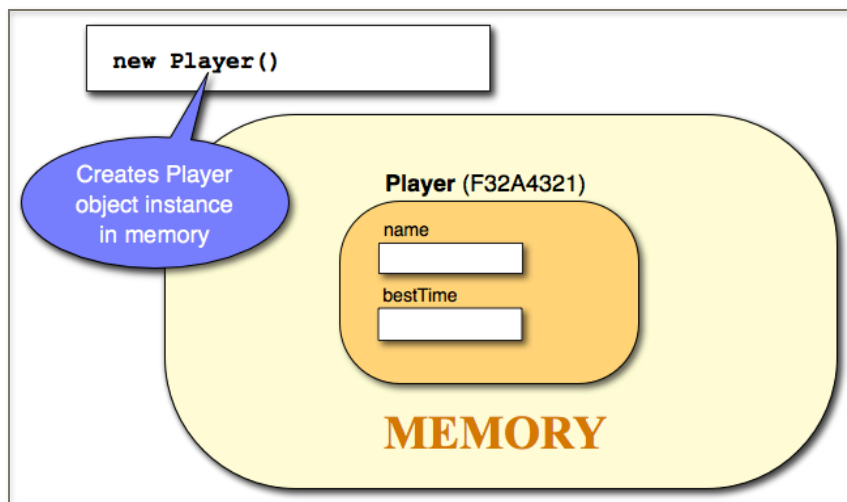
Creating objects from classes

A class describes a group of objects that all have the same variables definitions and methods. You use these classes to actually create the different object instances in your program. This is done by first specifying the new keyword followed by a call to the class's constructor as shown below.

```
new Classname()
```

Remember that the constructor is always the same name as the class.

Here is an example showing how to create a new `Player` object using the `Player` class datatype.

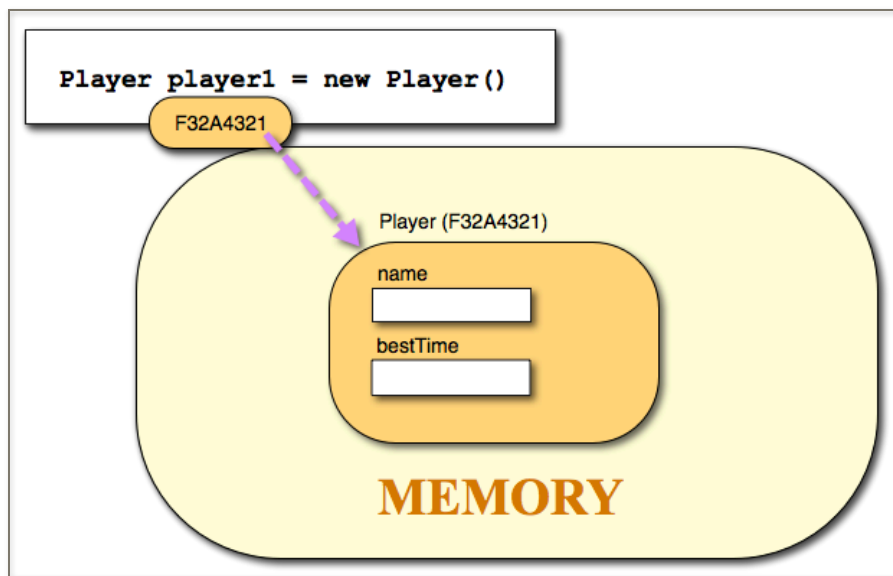


The `new` keyword tells Java that you are creating a new object. The `Player()` constructor is responsible for actually creating a new `Player` object instance in memory, and initializing its class variables with values. When the object is created, Java also creates memory locations for the `name` and `bestTime` class variables defined in the `Player` class. These two memory locations are associated with and belong to this specific object. This new `Player` object is located at the memory address `F32A4321`.

Defining Class Object Variables

Unfortunately, there is no way of directly accessing this new `Player` object and its `name` and `bestTime` variables unless you know its memory address. This is not very user-friendly.

Java solves this problem by allowing us to define a variable and then assign the memory address of the new object to the variable when the new object is created. This variable contains a reference or points to the new object just created as shown in the example below.



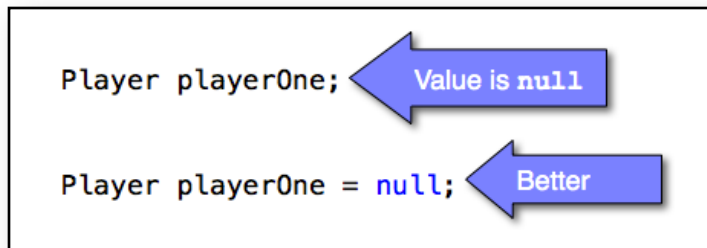
After the new `Player` object instance is created on the right, the memory address (`F32A4321`) of the new `Player` object instance is assigned to the `player1` variable on the left. The data type of the variable on the left must match the name of the class (`Player`) of the object created. The `player1` variable now contains a reference to or points to the new `Player` object instance created in memory.

You can easily access this new `Player` object and its contents by using the variable name (`player1`) that you assigned the object to. Specify the name of the variable followed by a dot (`.`), followed by the name of the attribute stored in the object instance. For example, to get the value of the `name` attribute in the `Player` object instance created earlier, you specify:

```
player1.name
```

The variable `player1` to the left of the dot (.) is used to qualify which object is being referenced. The value to the right of dot tells Java what in the object is to be accessed.

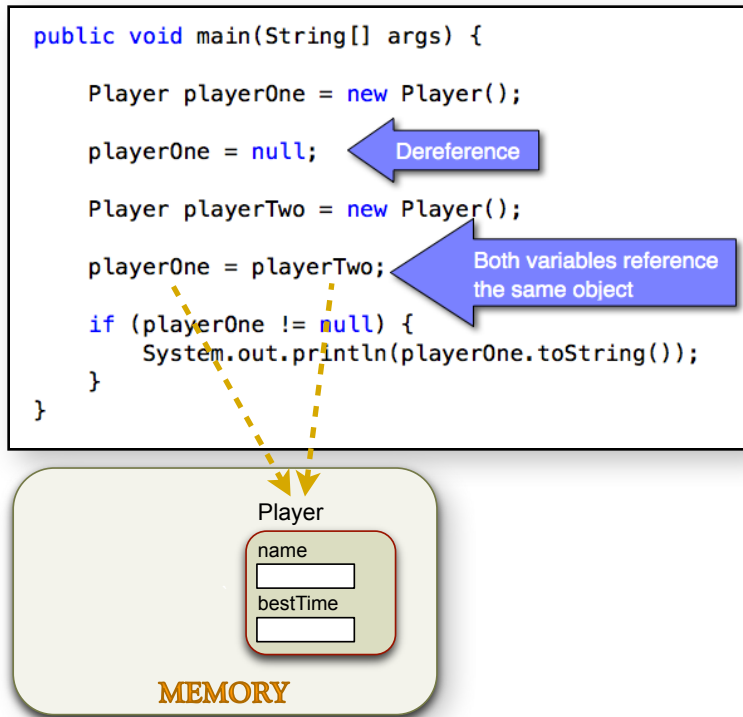
Sometimes we want to define class object variables before we create and assign an object instance to them as shown in the examples below.



The default value assigned to a class object variable is always `null`. The `null` keyword indicates that an object variable that it is not referencing any object. Both of the examples above are equivalent but the second is more explicit and is the preferred way of declaring a class object variable that has no value.

You can also dereference a class object variable at any time by assigning the `null` value to the variable. The variable no longer references or points to any object when the value is `null`. You can also assign a different object instance to a class object variable at any time. When this is done, the class object variable references the last object instance assigned to it.

This is illustrated in the example below.



A `null` value is assigned to the `playerOne` variable in the second statement. After this statement executes, it no longer is referencing the first `Player` object instance created in the statement above it.

We then created another new `Player` object instance and assigned it to the `playerTwo` class object variable. Finally, we assigned the value of the memory address stored in the `playerTwo` object to the `playerOne` class variable. Now both the `playerOne` and `playerTwo` class object variables reference the second `Player` object instance created in memory.

Java Bean Classes

A Java Bean is a special type of class in Java. It is typically used to implement each of the classes in the Model layer of an application. Each Java Bean class describes the data to be stored in each of the different types of objects that you want to model in your program. For example, you defined classes to store data the players, actors, items, map, locations, and scenes in the last lesson in your game in the UML Class Diagram. Each of these classes needs to be implemented as a Java Bean. Select the link below to view an introduction of Java Bean classes.

[View Introduction to Java Bean classes](#)

Each Java Bean class must abide by the following three basic conventions.

1. Implement the *Serializable* interface.

The *Serializable* interface allows the all of the data stored in class instance variables in a class to be translated into a format so that it can be stored on disk or sent across the network.

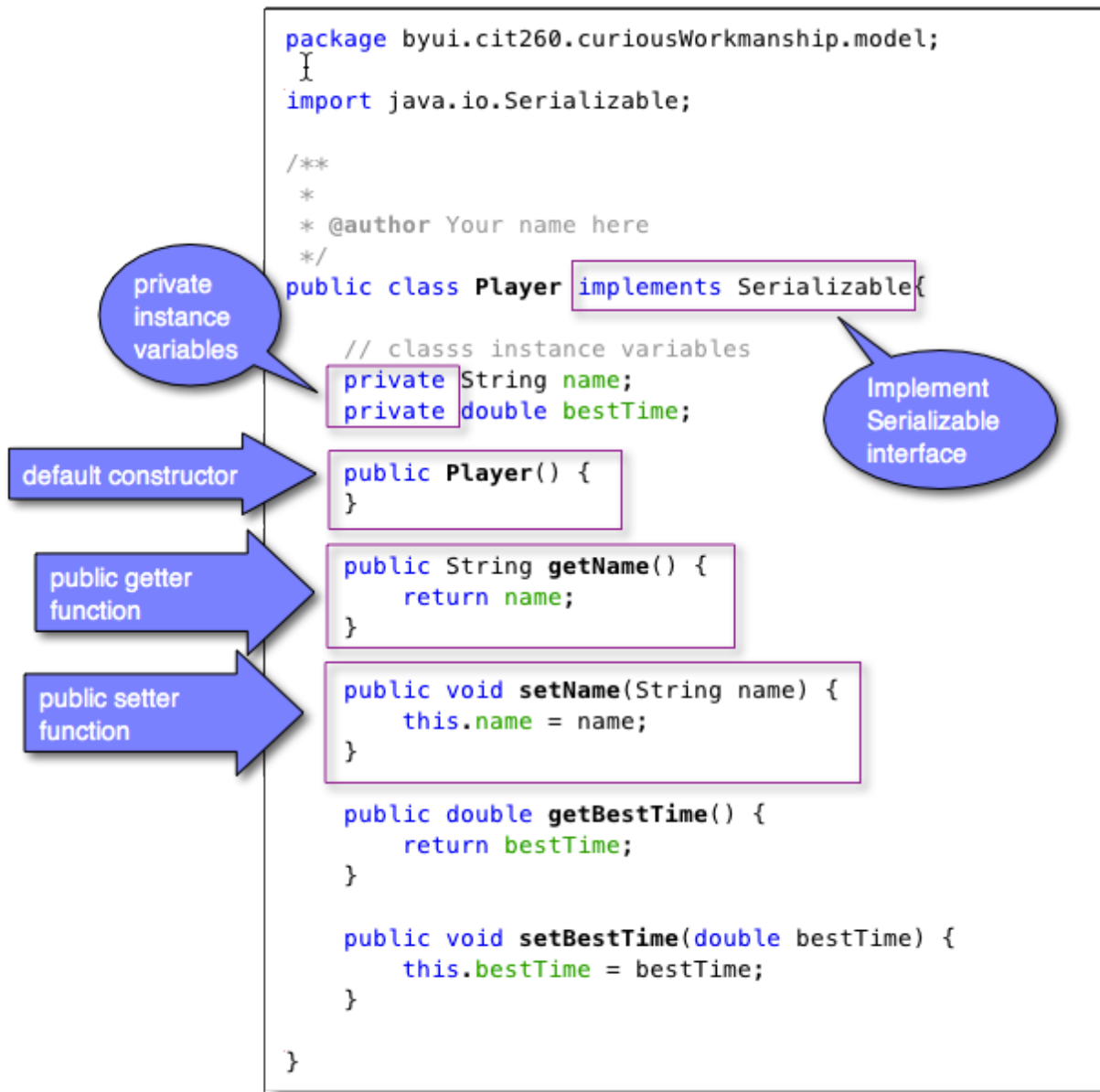
2. Create a default constructor for the class.

A constructor is used to create a new object from the class. A default constructor has no input parameters.

3. All class instance variables (attributes) should have the private modifier and have public “getter” and “setter” accessor methods.

It is also highly recommended that you implement the `toString()`, `equals()` and `hashCode()` methods. The `toString()` method (function) is useful for debug. It is used to print out the values of the class instance variables as a readable string. The `equals()` method is useful when you want to compare to see if the values of the instance variables in one object are equivalent to the value of the instance variables of another object of the same class. The `hashCode()` method speeds up sorting and searching for objects in a collection.

Here is an example of the `Player` Java Bean class. You make the class serializable by adding the keyword `implements` followed by the name of the *Serializable* interface after the name of the class. The default constructor method is normally added after the class attribute variables. The constructor name must match the class name. Finally, add the `private` modifier keyword in front of the type definition of each of the attribute variables, and add public getter and setter methods to the class for each of the class instance variables. The `toString()`, `equals()` and `hashCode()` methods have not been added in this example for simplicity.



Creating and using object instances

Select the link below to Watch the video below to learn how to create new instances of a class and to reference the instance variables and methods belonging to each of the objects.

[Watch the Creating and Using Objects Video](#) (Transcript)

The List Data Types

There are many times in programming that you need to model an ordered list of items. In the example game, we have a list of actors, a two-dimensional list of locations in the map, a list of the actors assigned to each location in the map, a list questions to be asked, a list of constant conversion values and a list of items stored in the warehouse.

Lists can have different characteristics. Some lists have a fixed length, such as the list of actors in the example game. Some lists vary in length, such as the list of items collected by an actor in the game. As the game progresses, items can be added or removed from the list. Some lists contain objects of different datatypes, while other lists contain objects all of the same type. Some lists are constant where the values cannot be changed.

Java provides three different data types to handle these different types of list: `array`, `ArrayList` and `Enum`.

Array An array is one of the native datatypes supported in most programming languages. An array is used to model a fixed length list of objects or primitive values (byte, short, int, long, float, double, boolean and char). All of the objects in the list must be of the same datatype. The elements assigned to each position in the list may be changed at any time. Each element in an array is referenced by an index.

ArrayList An `ArrayList` in Java is implemented as a class. It is a variable length list of objects. This means that you can dynamically add and remove objects from the list. The objects in the list may be of different datatypes. Primitive values (byte, short, int, long, float, double, boolean and char) cannot be directly put in an `ArrayList`. They must be converted into an object first.

Enum An `enum` a special type of class in Java. Like an array, it has a fixed length, and all of the elements in the list must be of the same datatype. It may be a list of objects or primitive values. Unlike an array, the elements stored in the array can not be changed. They are constant. The elements in an array are not referenced by an index. They are referenced by a keyword instead. The keyword maps to a specific element in the list.

When to use an array, enum, and ArrayList

An `ArrayList` is the most flexible of the three different types of lists so why not use them all the time? The answer to this question is that it depends on how the list is to be used.

Arrays and `enums` can be much more efficient to use from a performance perspective, because the number of items in a list is fixed. Memory is allocated for the entire list when the array or Enum list is initially created. With an `ArrayList`, memory is dynamically allocated each time an object is added or removed from the list. This is a very CPU intensive operation, so arrays and enums are more efficient to use when the length of list never changes.

Here are some guidelines to use when deciding what kind of a list to use in your program.

Guidelines to determine which kind of list to use

Array - When the length of the list is fixed and you may want to dynamically change the items in the list. All of the items in the list must be of the same datatype.

Enum - When the number items in the list are fixed and the items in the list will be constant and not change. All of the items in the list must be of the same datatype.

ArrayList - When you need to add or remove objects from the list, or the objects in the list need to be of different datatypes.

In the example game, we chose an array to model the list of inventory items in the game. All of the objects in the lists are of the same type (i.e., all `InventoryItem`). We chose an `ArrayList` to keep track of the objects collected for the ship because we will dynamically add and remove objects from the ship during the course of the game. Also, this list will contain objects of different datatypes. (e.g., `Barrel`, `Tool`, `Food`, etc.). We chose an `enum` list to store the list of actors (characters) in the game, because this is a fixed length list where the values assigned to each actor is constant and will never change.

Arrays

Arrays can be used to model one-dimensional list or multi-dimensional list. For example, an array can be used to implement a single list of grades or a two-dimensional table of values or objects, such as a spreadsheet. Arrays can also be used to model three, four or more dimensional list. Remember that arrays have a fixed length, and that all of the items in the list must be of the same data type.

One dimensional arrays

A one-dimensional array is used to model a single list of objects or primitive values that are all of the same data type.

Defining and creating one-dimensional arrays

You define an array variable done by first specifying the datatype of elements in the list followed by square brackets (`[]`) followed by the name of the variable.

dataType [] variableName;

Alternatively, you can also define an array by placing the square brackets after the variable name.

dataType variableName [] ;

```
double[] prices; // example of brackets after the data type
String names[]; // example of brackets after variable name
```

In either case, the square brackets are the symbols used to indicate an array in Java.

Defining a variable for an array does not actually create the array in memory. This is because an array is actually an object. You must first create the array using the `new` keyword followed by the datatype followed by the square brackets. You must specify the length of the array inside the square brackets.

new dataType [length]

Normally, the array is then assigned to an array variable of the same data type as shown below. The first five examples create arrays of primitive values. The last two examples create arrays of objects.

```
// Examples of defining one dimensional arrays of primitive values
char[] grades = new char[30];

int noOfStudents = 30;
long[] scores = new long[noOfStudents];
float[] percentages = new float[noOfStudents];
boolean[] passing = new boolean[noOfStudents];

// Examples of defining one dimensional arrays of objects
Student[] students = new Student[noOfStudents];
String[] assignments = new String[5];
```

Multi-dimensional arrays

One-dimensional arrays are used to model a single list of elements. Arrays can also be used to model multidimensional objects. For example, a two-dimensional array can be used to model a table of elements. A three-dimensional array can be used to model a cube of elements, and so on.

Two-dimensional arrays are used so often in programming that it is worth spending more time learning about how to implement and use dimensional arrays. For example in our game program, we use a two-dimensional array to store all of the `Locations` in the `Map`.

Defining and creating a two-dimensional array

Defining a variable for a two-dimensional array is very similar to defining an array for a single dimensional array, except that you add an additional set of square brackets to indicate the second dimension.

dataType *variableName* [] []

You create a two-dimensional array by using the `new` keyword, followed by the data type of each element in the array, followed by the number of rows in the array enclosed in square brackets, followed by the number of columns in the array enclosed in square brackets.

`new dataType [noOfRows] [noOfColumns]`

Here are several examples of defining and creating two-dimensional arrays.

```
// Examples of defining and creating two dimensional arrays of objects
String[][] quantities;
quantities = new String[5][10];

double[][] penalties = new double[3][4];

int noOfRows = 8;
int noOfColumns = 10;
Location[][] locationsInMap = new Location[noOfRows][noOfColumns];
```

In the first example, we defined the variable, `quantities`, as a two-dimensional array of `String` objects. Then we created a two-dimensional array of 5 rows and 10 columns and assigned it to the variable `quantities`. In the second example, the definition of the `penalties` variable and, and creation a two-dimensional array of 3 rows and 4 columns of `double` values are combined into a single statement. In the last example, we used variables instead of literal values to define the number of rows and columns in the array to define a two-dimensional array of `Location` objects instance.

ArrayList

An `ArrayList` is one of the classes that comes shipped with Java. An `ArrayList` is a good choice to use when you need to dynamically add or remove elements to a list in your program. An `ArrayList` object does not have a fixed length.

Defining ArrayList

`ArrayList` variables are defined by first specifying the class name, `ArrayList`, and followed by the datatype of the elements in the list enclosed within the `< >` characters followed by the name of the variable.

`ArrayList<dataType> variableName`

```
ArrayList<Weapon> weapons
```

An `ArrayList` is created using the `new` keyword, followed by the name of the class, `ArrayList`, optionally, followed by the data type enclosed within the `< >` followed by parenthesis.

```
new ArrayList<dataType>()
```

```
new ArrayList<Actor>()
```

You may create a new `ArrayList` object in memory, and assign it to a newly defined variable in one statement as shown below. Notice that you do not need to specify the data type of the elements within the `< >` on the right-hand side when you create the `ArrayList` object, because Java can infer the data type of the elements in the list from the variable definition on the left.

```
// Examples of defining ArrayList
ArrayList<Actor> actors = new ArrayList<>();
ArrayList<Item> inventoryItems = new ArrayList<>();
```

Enum

The third type of list in Java is an enum. An enum list is similar to an array in that it has a fixed length list and all of the items in the list must be of the same data type. A enum differs from an array in that the items in the list are constant or final. The values associated with each item in the list can never change. In addition, the items in the enum list are not addressed by an index. Instead, each item in the list is mapped to a keyword. You address or locate an item in the list by its associated keyword.

In the example game, we have several fixed list that will never change during the course of the game. The list of actors (characters) in the game, the directions on the compass (i.e., North, South, East, West) and the list of recommended amounts for each resource to be collected are all fixed length list whose values will never change.

Defining enum classes

An enum list is a special type class. It can be used to define a simple list of integer values or a list of objects that have class variables where the values of the variables are constant or fixed.

A simple enum list

A simple enum list contains only a list of keywords. Assigned to each keyword is its ordinal value or position in the list. The first keyword in the list has a value of 0, the next 1, the next 2 and so on. This type of list is often used to associate a keyword with an index position of an item in an array or ArrayList.

You always start an enum list by specifying the enum keyword followed by a pair of curly braces { } to indicate the start and end of the enum class. Inside the curly braces, you list all of the keywords separated by commas. The last keyword must be followed by a semicolon (;).

```
enum classname {  
    keyword1,  
    keyword2,  
    keyword3,  
    ...  
    keyword;  
}
```

Here is an example of a simple enum class where the ordinal values 0, 1, 2, 3 are automatically assigned to the keywords, `regular_scene`, `knowledge_scene`, `gather_scene` and `warehouse_scene`. The data type of the ordinal value associated with each these keywords is always an `int`.

```
public enum SceneType {  
    regular_scene,  
    knowledge_scene,  
    gather_scene,  
    warehouse_scene  
}
```

You access the value of the ordinal position associated with each keyword by calling the enum class's `ordinal()` method (function) using the following syntax:

```
EnumClassName.keyword.ordinal()
```

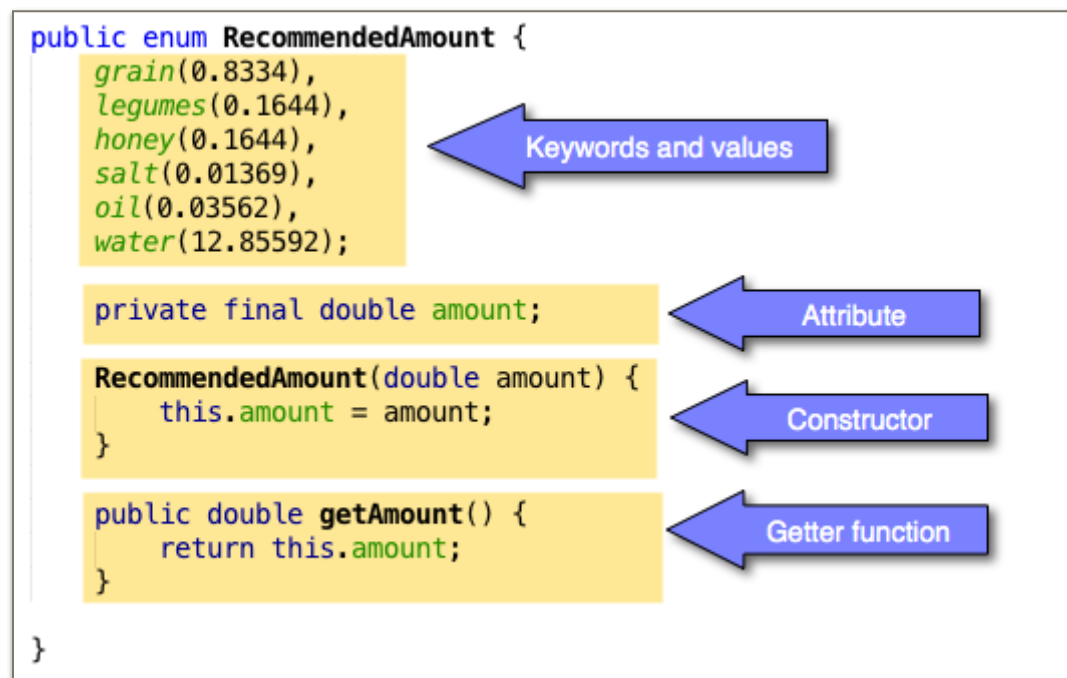
For example, to get the ordinal position of the `gather_scene` keyword in the `SceneType` enum, you would write the following.

```
SceneType.gather_scene.ordinal()
```

Defining an enum class of fixed objects

At other times, you may need to create a fixed list of objects that have one or more class data variables associated with each object. A unique keyword is assigned to each object in the list. Each keyword still has an ordinal value specifying its position in the list, but it also will have an actual object instance assigned to it. Each of these objects will contain the fixed values that are assigned to each of the class data variables that belong to that specific enum object.

For example, we created an enum class in the example game to define the recommended amount to be stored for each item on the ship.



You start an enum class by specifying the `enum` keyword, followed by the name of the enum class followed by curly braces. Inside the curly braces, you must define the keywords to be associated with each enum object in the list. You can optionally specify a list of constant values in parenthesis after the keyword to be passed to and assigned to the class attributes when each enum object is created.

The class attributes are normally defined after the keywords. The `private` `final` modifiers should be specified in front of the data type. The `final` modifier indicates that this attribute is a constant and that the value assigned to it is final and can never be changed.

The constructor generally is defined after the class attributes. It is automatically called to create and associate an enum object with each keyword in the list. It must assign a constant value to each of the class attributes. Optionally, you can define a list of parameters variables that will be used to assign a unique constant value to a corresponding class attribute.

In the `RecommendedAmount` enum class above, the constant value in parenthesis after each keyword is passed to constructor's `amount` parameter variable and then assigned to the `amount` class attribute for the enum object associated with that keyword. For example, the value stored in the `amount` variable for the “grain” object will be `0.8344`, the value of stored in the “legumes” object will be `0.1644` and so on.

Finally, you define “getter” methods for each of the class attributes. These methods can be used to get a constant value assigned to the class attributes in a specific enum object. You do not create “setter” methods for the class attributes because the values of the class attributes can never be changed.

Here is the syntax for calling a “getter” method (function) to get the value of a class variable for one specific enum object.

```
EnumClassName.keyword.getVariablename()
```

Here is an example of how to call the `getAmount()` “getter” method retrieve the value stored in the `amount` class attribute for the “honey” enum object in the example above.

```
RecommendedAmount.honey.getAmount()
```

Conclusion

You have learned about how to define primitive variables, Java Bean classes and the different types of lists in Java. Now it is time to put this into practice. Proceed on to the team assignment for this lesson.