

# Chapter 14

## Classes 1

---

### Java AP

**Copyright Notice**

Copyright © 2013 DigiPen (USA) Corp. and its owners. All Rights Reserved

No parts of this publication may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language without the express written permission of DigiPen (USA) Corp., 9931 Willows Road NE, Redmond, WA 98052

**Trademarks**

DigiPen® is a registered trademark of DigiPen (USA) Corp.

All other product names mentioned in this booklet are trademarks or registered trademarks of their respective companies and are hereby acknowledged.

## Introduction to Object-Oriented Programming

- Object-oriented programming (OOP) is a way of organizing the code in a program by grouping it into objects.
- Objects, are individual elements that include information (data values and functionality).
- Using an object-oriented approach to organizing a program allows you to group particular pieces of information (for example, music information like album title, track title, or artist name) together with common functionality or actions associated with that information (such as “add track to playlist” or “play all songs by this artist”). These items are combined into a single item, an object (for example, an “Album” or “MusicTrack”).
- For example, if we want to create a **player** class for our game we would have the following data fields and functionalities added to it:

Class Name	Player
Data Fields	<ul style="list-style-type: none"> <li>➤ Shape</li> <li>➤ Position</li> <li>➤ Scale</li> <li>➤ Rotation</li> <li>➤ Speed</li> <li>➤ Health</li> <li>➤ etc...</li> </ul>
Functionalities (Methods)	<ul style="list-style-type: none"> <li>➤ Run</li> <li>➤ Jump</li> <li>➤ Shoot</li> <li>➤ Play Animation</li> <li>➤ etc...</li> </ul>

- Now, in our code we can create one or more **objects** of type "**Player**" (player1, player2, etc...) and each one of them will have all the above fields and functionalities.

**Note:** A "**Player object**" is an "**instance of type Player**". You have one class "**Player**" but can create multiple objects or instances of type **Player**.

## Procedural programming vs Object-Oriented programming

Procedural Programming	Object- Oriented Programming
<ul style="list-style-type: none"> <li>• There is a distinct division between code (functions) and data.</li> <li>• Data is passive, it gets acted upon by functions.</li> <li>• We generally pass the data between functions.</li> </ul>	<ul style="list-style-type: none"> <li>• Code and data are encapsulated together in a single object.</li> <li>• The functions that work on the data are part of the object.</li> <li>• We don't have to pass data to the functions.</li> </ul>

Usually, for a language to be considered object-oriented, it should have these three properties (which we will cover in depth in this and future chapters) :

1. Encapsulation (data abstraction/hiding)
2. Inheritance (relationships between entities)
3. Polymorphism (runtime decisions)

Java is designed to be used in a purely object-oriented manner. So far we have been using static to try and recreate the behavior of a procedural language. In a language like C/C++ these functions would be global, i.e they do not belong to any class. As Java is a purely object-oriented language all data and methods must exist in a class.

## Classes and Objects

- A class is an abstract representation of an object. It stores information about the types of data that an object can hold and the behaviors that an object can exhibit.
- When creating a class you are creating your custom type that contains other types and functions.
- The usefulness of such an abstraction may not be apparent when you write small scripts that contain only a few objects interacting with one another.
- As the scope of a program grows, however, and the number of objects that must be managed increases, you may find that classes allow you to better control how objects are created and how they interact with one another.

The general syntax of a **class** in Java:

```

modifier class ClassName
{
    /* Data Fields Declarations */

    /* Method Declarations */
}

```

Declaration	Purpose
modifier	Used to define the class accessibility. So far we've been using "public" which means that the class is accessible by all other classes. We will learn more about modifiers in future chapters.
class	The class keyword tells the compiler that we are declaring a class.
ClassName	A unique name for the class that we are creating.
Data Declarations	Optional data fields that the class instances will have.
Method Declarations	Optional functionalities that the class instances will have.

**Note:** *A function that belongs to a class is called a "Method".*

Example	<pre> public class Player {     /* Data Fields Declarations */     public float positionX;     public float positionY;     public int health;      /* Method Declarations*/     public void Initialize()     {         positionX = 10.0f;         positionY = 50.0f;         health = 100;     } } </pre>
---------	---

## Creating a class

Every class should be created in its own file. You cannot have more than one class in a file and the file name should be the same as the class name but with a ".java" extension.

File Name	Code	Explanation
<b>Player.java</b>	<pre>public class Player {     /* Data Fields Declarations */      /* Method Declarations*/ }</pre>	Valid class creation: <ul style="list-style-type: none"> <li>File contains on class only</li> <li>File name matches Class name</li> </ul>
<b>Foo.java</b>	<pre>public class Player {     /* Data Fields Declarations */      /* Method Declarations*/ }</pre>	Invalid class creation: <ul style="list-style-type: none"> <li>File name doesn't match class name</li> <li>Error shown:</li> </ul> <pre>Foo.java:1: error: class Player is public, should be declared in a file named Player.java public class Player       ^</pre>
<b>Player.java</b>	<pre>public class Player {     /* Data Fields Declarations */      /* Method Declarations*/ }  public class Foo { }</pre>	Invalid class creation: <ul style="list-style-type: none"> <li>File contains more than one class</li> <li>To fix it, move the Foo class into a separate file called Foo.java</li> <li>Error Shown:</li> </ul> <pre>Player.java:8: error: class Foo is public, should be declared in a file named Foo.java public class Foo       ^</pre>

**Note:** Notice that all classes names start with a capital letter, this is not necessary however does follow the published guidelines for Java naming conventions:

<http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html#367>

## Instantiating a class

Now that we know how to create a class, we need to learn how to create instances (a.k.a objects) of that class. In Java, we create an instance of a class by using the **new** keyword.

<p>Example</p>	<pre> /* File: Player.java */ public class Player {     /* Data Fields Declarations */     public float positionX;     public float positionY;     public int health;      /* Class Constructor */     public Player()     {         positionX = 10.0f;         positionY = 50.0f;         health = 100;     } } </pre>
<p>Explanation</p>	<pre> /* File: Main.java */ public class Main {     public static void main(String[] args)     {         /* Creating an instance of type Player */         Player p = new Player();     } } </pre> <p>Inside the main function, we are creating one instance of the <b>Player</b> class which will be referred to by the variable "<b>p</b>". In order to create that instance, we had to:</p> <ul style="list-style-type: none"> <li>• Use the <b>new</b> operator, which will create the object.</li> <li>• Followed by a call to the <b>class constructor</b> (will be explained in details in the next section), which initializes the newly created object.</li> </ul>

## Constructors

- A constructor is a function inside a class that has the **same name as the class**.
- It will be called **only once**, when you create an object of type that class.
- The constructor can take parameters but can **never have a return type**(since by default it should return an object of that class' type).
- Usually the constructor is used to **initialize the data fields of a class**.
- If no constructors are defined, Java will add a **default constructor** to the class. That default constructor will not initialize the class data fields though.
- A class can contain more than one constructor:

### Example

```
/* File: Player.java */
public class Player
{
    /* Data Fields Declarations */
    public float positionX;
    public float positionY;
    public int health;

    /* Default Constructor */
    public Player()
    {
        positionX = 0.0f;
        positionY = 0.0f;
        health = 100;
    }

    /* Constructor 2 */
    public Player(float positionX_, float positionY_)
    {
        positionX = positionX_;
        positionY = positionY_;
        health = 100;
    }
}
```

```

/* File: Main.java */
public class Main
{
    public static void main(String[] args)
    {
        /* Using the default constructor */
        Player p = new Player();

        /* Using constructor 2 */
        Player p2 = new Player(50.0f, 100.0f);
    }
}

```

**Explanation**

The first Player instance "**p**" is created using the default constructor which means that the values of **positionX** and **positionY** inside it are equal to 0.

The second Player instance "**p2**" is created using the second constructor which means that the values of **positionX** and **positionY** inside it are equal to the values specified by the user through the constructor's parameters **positionX\_** and **positionY\_** (in this case 50 and 100).

***Note: If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments. Based on the arguments the Java compiler knows which constructor to call.***



Earlier it was mentioned that the compiler will generate a default constructor for you. However this only holds true if you did not specify any custom constructor(s). This behavior allows you to create a class that cannot be constructed without data being passed into it.

### Example

```
/* File: Player.java */
public class Player
{
    /* Data Field Declarations */
    public float positionX;
    public float positionY;
    public int health;

    /* Custom Constructor */
    public Player(float positionX_, float positionY_)
    {
        positionX = positionX_;
        positionY = positionY_;
        health = 100;
    }
}
```

```
/* File: Main.java */
public class Main
{
    public static void main(String[] args)
    {
        /* Attempting to use a default constructor */
        Player p = new Player();

        /* Using the custom constructor */
        Player p2 = new Player(50.0f, 100.0f);
    }
}
```

### Compiler Error

```
Exception in thread "main" java.lang.Error: Unresolved
compilation problem:
    The constructor Player() is undefined

    at Main.main(Main.java:7)
```

The only way to create a **Player** instance is by using the custom constructor.

## Accessing the instance fields

Once you use the constructor and instantiate an object, you have the ability to access the data fields of that object using the **dot "." operator**.

Example	<pre> /* File: Player.java */ public class Player {     /* Data Field Declarations */     public float positionX;     public float positionY;     public int health;      /* Constructor */     public Player(float positionX_, float positionY_)     {         positionX = positionX_;         positionY = positionY_;         health = 100;     } } </pre>
Output	<pre> /* File: Main.java */ public class Main {     public static void main(String[] args)     {         Player p = new Player(50.0f, 100.0f);         System.out.println("Position X: " + p.positionX);         System.out.println("Position Y: " + p.positionY);         System.out.println("Health: " + p.health);     } } </pre>
Explanation	<p>Position X: 50.0 Position Y: 100.0 Health: 100</p> <p>Using the "." operator we were able to access all the instance's fields (<i>in this case all fields are public which allowed us to access them. We will be covering that topic in much more details in a future chapter</i>).</p>

Every instance created will contain its own copy of the class data fields.

### Example

```
/* File: Player.java */
public class Player
{
    /* Data Fields Declarations */
    public float positionX;
    public float positionY;
    public int health;

    /* Constructor */
    public Player(float positionX_, float positionY_)
    {
        positionX = positionX_;
        positionY = positionY_;
        health = 100;
    }
}
```

```
/* File: Main.java */
public class Main
{
    public static void main(String[] args)
    {
        Player p = new Player(50.0f, 100.0f);
        System.out.println("p Position X: " + p.positionX);
        System.out.println("p Position Y: " + p.positionY);
        System.out.println("p Health: " + p.health);

        Player p2 = new Player(150.0f, 400.0f);
        System.out.println("p2 Position X: " + p2.positionX);
        System.out.println("p2 Position Y: " + p2.positionY);
        System.out.println("p2 Health: " + p2.health);
    }
}
```

### Output

```
p Position X: 50.0
p Position Y: 100.0
p Health: 100
p2 Position X: 150.0
p2 Position Y: 400.0
p2 Health: 100
```

**Explanation**

As you can see, **p** has a totally different field values than **p2**. They are totally two independent instances and their fields do not share memory.

**Encapsulation**

Now that we know what classes are and how we can use them, it is time to understand what **encapsulation** is and why it is one of the most important features that an object oriented language provides.

Without classes, creating two players with the same information as the example above would be done in the following manner:

**Example**

```
/* File: Main.java */
public class Main
{
    public static void main(String[] args)
    {
        float p1PositionX = 50.0f;
        float p1PositionY = 100.0f;
        int p1Health = 50;

        System.out.println("P1 Position X: " + p1PositionX);
        System.out.println("P1 Position Y: " + p1PositionY);
        System.out.println("P1 Health: " + p1Health);

        float p2PositionX = 150.0f;
        float p2PositionY = 400.0f;
        int p2Health = 100;

        System.out.println("P2 Position X: " + p2PositionX);
        System.out.println("P2 Position Y: " + p2PositionY);
        System.out.println("P2 Health: " + p2Health);
    }
}
```

Without classes and objects coding is messy and hard to keep track of, but by making a class and using it to create instances that contain those fields we make the code easier to manage and to read. Adding data fields and methods inside a class is known as **encapsulating** data inside a class.

If we want to fully encapsulate the above example we would write our **Player** class as follows:

<p><b>Example</b></p>	<pre> /* File: Player.java */ public class Player {     /* Data Fields Declarations */     public float positionX;     public float positionY;     public int health;      public Player(float positionX_, float positionY_)     {         positionX = positionX_;         positionY = positionY_;         health = 100;     }      public void PrintInfo()     {         System.out.println("Position X: " + positionX);         System.out.println("Position Y: " + positionY);         System.out.println("Health: " + health);     } } </pre> <pre> /* File: Main.java */ public class Main {     public static void main(String[] args)     {         Player p = new Player(50.0f, 100.0f);         p.PrintInfo();          Player p2 = new Player(150.0f, 400.0f);         p2.PrintInfo();     } } </pre>
<p><b>Output</b></p>	<pre> Position X: 50.0 Position Y: 100.0 Health: 100 Position X: 150.0 Position Y: 400.0 Health: 100 </pre>

**Note:** All methods inside a class have access to the class data fields. You can see that both the constructor and the "PrintInfo" methods are accessing the "positionX", "positionY" and "health" fields.

## The *this* reference

Notice how, in the above example, we were able to encapsulate the ***PrintInfo()*** function and depending on which instance calls the function the values of all the fields change accordingly. But how does the ***PrintInfo()*** function know which instance called it? Was it ***p1*** or ***p2***?

Well, let me reveal the secret. Every method call gets an ***implicit parameter*** which is a ***reference to the instance that called the method***. We call that implicit parameter the ***this reference***.

You can actually use the ***this reference*** inside all class methods:

### Example

```
/* File: Player.java */
public class Player
{
    /* Data Fields Declarations */
    public float positionX;
    public float positionY;
    public int health;

    public Player(float positionX_, float positionY_)
    {
        this.positionX = positionX_;
        this.positionY = positionY_;
        this.health = 100;
    }

    public void PrintInfo()
    {
        System.out.println("Position X: " + this.positionX);
        System.out.println("Position Y: " + this.positionY);
        System.out.println("Health: " + this.health);
    }
}
```

Again, the ***this reference*** is implicitly used by the method so no need to actually use it all the time.

On the other hand, the **this reference** can be very helpful if we would like to call a class' constructor inside another one of the class' constructors.

**Example**

```
/* File: Player.java */
public class Player
{
    /* Data Declarations */
    public float positionX;
    public float positionY;
    public int health;

    public Player()
    {
        this(0,0);
        System.out.println("End Of Default Constructor");
    }

    public Player(float positionX_, float positionY_)
    {
        positionX = positionX_;
        positionY = positionY_;
        health = 100;
        System.out.println("End Of Custom Constructor");
    }

    public void PrintInfo()
    {
        System.out.println("Position X: " + positionX);
        System.out.println("Position Y: " + positionY);
        System.out.println("Health: " + health);
    }
}
```

```
/* File: Main.java */
public class Main
{
    public static void main(String[] args)
    {
        Player p = new Player();
        p.PrintInfo();
    }
}
```

<b>Output</b>	End Of Custom Constructor End Of Default Constructor Position X: 0.0 Position Y: 0.0 Health: 100
---------------	--

**Note:** *Calling the constructor using the this reference has to be the first statement in the calling constructor otherwise we will get a compiler error.*

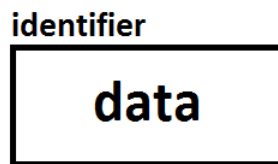
## References

### Recap on Primitive Data Types VS Reference Data Types

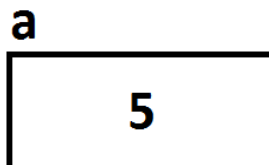
In Java we have two kind of data types: **primitive data types** and **reference data types**.

A primitive type is predefined by the language and is named by a reserved keyword.

*PrimitiveType identifier = data;*

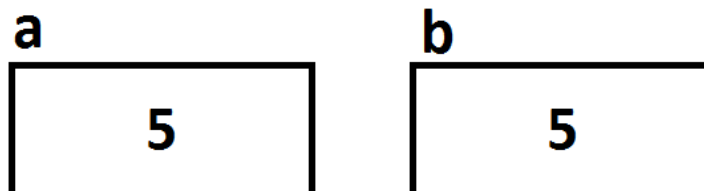


for example: `int a = 5;`



Primitive values do not share state with other primitive values.

for example: `int a = 5;`  
`int b = a;`





The eight primitive data types supported by the Java programming language are: **byte**, **short**, **int**, **long**, **float**, **double**, **boolean** and **char**.

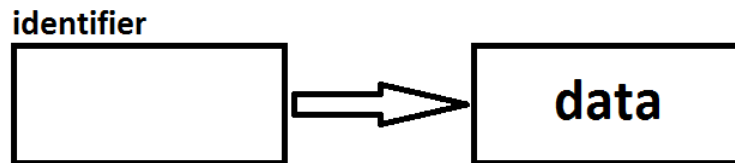
Example	<pre>public class Main {     public static void main(String[] args)     {         int a = 5;         int b = a;          System.out.println("a = " + a);         System.out.println("b = " + b);          a = 6;         System.out.println("a = " + a);         System.out.println("b = " + b);          b = 7;         System.out.println("a = " + a);         System.out.println("b = " + b);     } }</pre>
Output	<pre>a = 5 b = 5 a = 6 b = 5 a = 6 b = 7</pre>

As you can see, even though we said "**b = a**" the two variables stayed separate. Changing the value of **a** doesn't affect **b** and vice versa.

**Note:** *In addition to the eight primitive data types listed above, the Java programming language also provides special support for strings (java.lang.String). String is not a primitive data type, it is an immutable object (which means that once created, their values cannot be changed). Immutable objects can be thought of as primitive types since they behave a lot like one. We will cover the String object in much more details in a future chapter.*

However, a reference data type (a.k.a object) is only a means to access the data.

*ReferenceType identifier = data;*



### Example

```

/* File: Player.java */
public class Player
{
    /* Data Fields Declarations */
    public float positionX;
    public float positionY;
    public int health;

    public Player(float positionX_, float positionY_)
    {
        positionX = positionX_;
        positionY = positionY_;
        health = 100;
    }

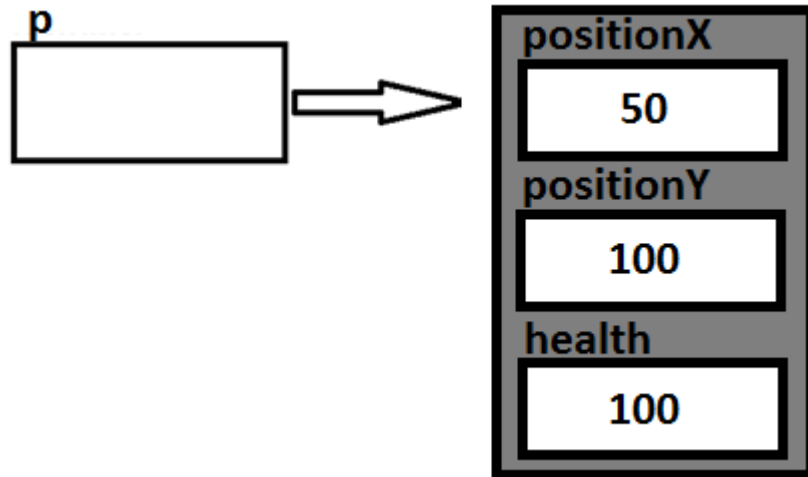
    public void PrintInfo()
    {
        System.out.println("Position X: " + positionX);
        System.out.println("Position Y: " + positionY);
        System.out.println("Health: " + health);
    }
}
  
```

```

/* File: Main.java */
public class Main
{
    public static void main(String[] args)
    {
        Player p = new Player(50.0f, 100.0f);
        p.PrintInfo();
    }
}
  
```

**Explanation**

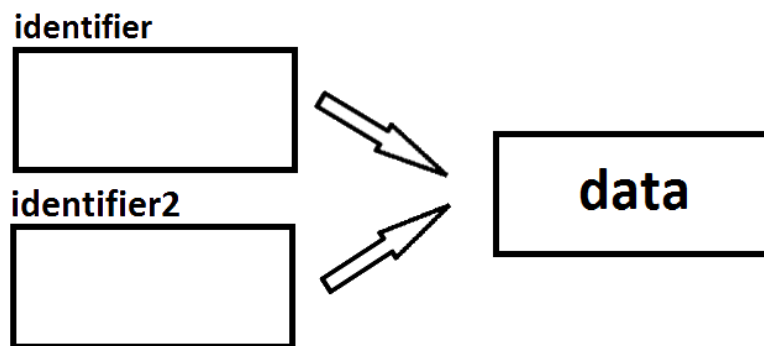
**p** is a reference to a **Player** object which means that through **p** we can access the **Player** memory /data that was allocated by the **new** operator and initialized by the **Player** class constructor.



Suppose the following declaration is made:

```
ReferenceType identifier = data;
```

```
ReferenceType identifier2 = identifier;
```



Both variables are referencing the same data, so if any of them change the data then it is changed for both.

## Example

```
/* File: Player.java */
public class Player
{
    /* Data Fields Declarations */
    public float positionX;
    public float positionY;
    public int health;

    public Player(float positionX_, float positionY_)
    {
        positionX = positionX_;
        positionY = positionY_;
        health = 100;
    }

    public void PrintInfo()
    {
        System.out.println("Position X: " + positionX);
        System.out.println("Position Y: " + positionY);
        System.out.println("Health: " + health);
    }
}
```

```
/* File: Main.java */
public class Main
{
    public static void main(String[] args)
    {
        Player p = new Player(0.0f, 0.0f);
        System.out.println("p content:");
        p.PrintInfo();
        System.out.println();

        Player p2 = p;
        System.out.println("p2 content:");
        p2.PrintInfo();
        System.out.println();

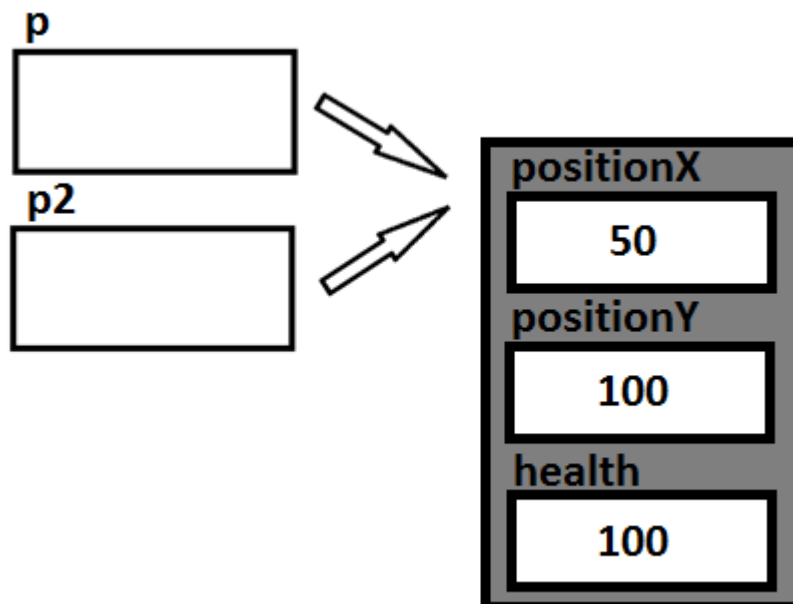
        p.positionX = 50;
        p.positionY = 100;

        System.out.println("p content:");
        p.PrintInfo();
        System.out.println();

        System.out.println("p2 content:");
        p2.PrintInfo();
    }
}
```

**Output**

```
p content:  
Position X: 0.0  
Position Y: 0.0  
Health: 100  
  
p2 content:  
Position X: 0.0  
Position Y: 0.0  
Health: 100  
  
p content:  
Position X: 300.0  
Position Y: 0.0  
Health: 100  
  
p2 content:  
Position X: 300.0  
Position Y: 0.0  
Health: 100
```

**Explanation**

Since both **p** and **p2** are referencing the same **data**, if one changes a field's value it is changed for both.

**Note:** Having two references for the same data is known as *aliasing*. Aliasing can cause unintended problems for the programmer.

### The Null Reference

Any reference type variable that is declared without being initialized is called a *null reference* or *null pointer*. In other words, a null reference is when the reference variable is created but is pointing to nothing.

*ReferenceType identifier;*

identifier

**null**

You can always check if the reference variable is initialized or not.

```
if ( identifier != null) /* true means identifier is initialized */
```

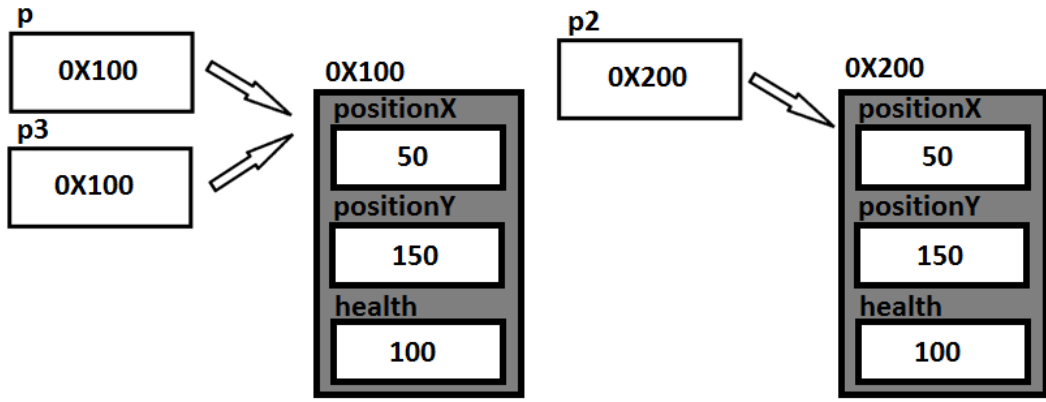
**Note:** An attempt to access data through a null reference will cause your program to terminate with a *NullPointerException*.

Example	<pre>public class Main {     public static void main(String[] args)     {         Player p;         p.positionX = 300;     } }</pre>
Output	<pre>Exception in thread "main" java.lang.Error: Unresolved compilation problem:     The local variable p may not have been initialized      at Main.main(Main.java:10)</pre>
Explanation	<p><b>p</b></p> <p><b>null</b></p> <p>Like all <b>reference types</b>, if not initialized using the <b>new operator</b> and the <b>class constructor</b> its content is <b>null</b>. Trying to access fields though a null reference will lead to a crash because the reference is not pointing to any data.</p>

## Comparing References

When comparing references, you are comparing addresses and not data field values.

<b>Example</b>	<pre> /* File: Player.java */ public class Player {     /* Data Fields Declarations */     public float positionX;     public float positionY;     public int health;      public Player()     {         positionX = 0;         positionY = 0;         health = 100;     } } </pre>
<b>Example</b>	<pre> /* File: Main.java */ public class Main {     public static void main(String[] args)     {         Player p = new Player();         p.positionX = 50;         p.positionY = 150;          Player p2 = new Player();         p2.positionX = 50;         p2.positionY = 150;          Player p3 = p;          System.out.println("p == p2: " + (p == p2));         System.out.println("p == p3: " + (p == p3));     } } </pre>
<b>Output</b>	<pre> p == p2: false p == p3: true </pre>
	<p>Even though all three Player references have the same values in their fields, when comparing two references of the same type we are actually comparing the addresses they are pointing to.</p>

Explanation	 <p>p == p2 is actually doing 0X100 == 0X200 which results to a false. p == p3 is actually doing 0X100 == 0X100 which results to a true.</p>
-------------	--

## Packages

We previously learned how to create a class but one important part was omitted, **packaging a class**. In Java, the packaging feature allows us to organize our code. Classes that belong to the same category are put in the same package. For example, if we build math related classes we can package them all in a math package.

Example	<pre> /* File: Vector2D.java */ package math;  public class Vector2D {     public float x;     public float y; } </pre> <hr/> <pre> /* File: Vector3D.java */ package math;  public class Vector3D {     public float x;     public float y;     public float z; } </pre>
---------	---

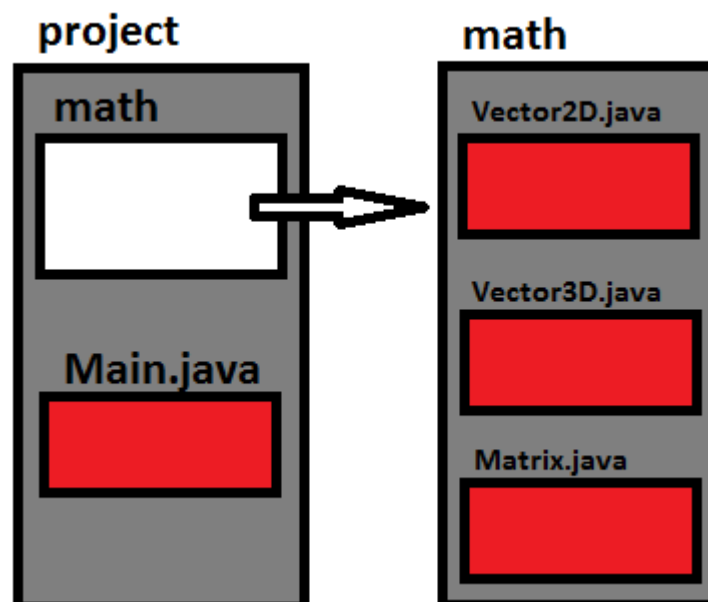


```
/* File: Matrix.java */  
package math;  
  
public class Matrix  
{  
    public float m[][];  
}
```

```
/* File: Main.java */  
  
import math.Vector2D;  
  
public class Main  
{  
    public static void main(String[] args)  
    {  
        Vector2D v = new Vector2D();  
        v.x = 10.0f;  
        v.y = 100.0f;  
    }  
}
```

**Explanation**

As mentioned above, in order to organize a project we put classes that belong to the same category under the same package, which means the classes files will be in the same folder. Every file created inside a specific folder needs to be packaged accordingly. For example, the Vector2D class is created inside a "math" folder in our project folder. The same goes for the Vector3D and Matrix classes.



Specifying the package inside a class is simply done by adding "**package** foldername;" at the top of the class.

Classes created in different packages can't access each other unless the class is **imported**. The **Main** class had to import the **Vector2D** class before creating a **Vector2D** instance.

You can see that above the class declaration in the Main.java file:

```
import math.Vector2D;
```

If the Vector2D class wasn't imported, the Vector type wouldn't have been recognized by the compiler in this file and we would have gotten the following error:

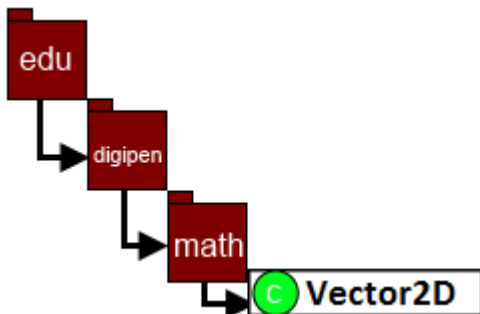
```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
  Vector2D cannot be resolved to a type
  Vector2D cannot be resolved to a type

  at Main.main(Main.java:9)
```

**Note:** *Classes that belong to the same package don't need to import each other.*

### Multiple folders

Of course, when organizing files we will end up with folders inside folders, for example:



Packaging the **Vector2D** class: **package** edu.digipen.math;

Importing the **Vector2D** class: **import** edu.digipen.math.Vector2D;

**Note:** *It is standard practice to leave the package name completely lowercase. For some uses of Java, Android for example ignoring this can lead to obscure errors. This also prevents collision with the names of classes which should start with a capital letter.*

**Importing all classes in a package**

It is possible to import all the package's classes with one import by using the asterisk symbol (\*)

```
import math.*;
```

That will import Vector2D, Vector3D and Matrix all at once.

**Using fully qualified package names**

The below example assumes we have a **Vector** class inside a **math** package.

<b>Example</b>	<pre>public class Main {     public static void main(String[] args)     {         math.Vector v = new math.Vector();          java.util.Vector vec = new java.util.Vector(3, 2);     } }</pre>
<b>Explanation</b>	<p>Instead of importing a class, you can use the fully qualified package name instead in order for the compiler to reach that class and know what type you want.</p> <p>The fully qualified name is useful when you want to use two classes from two packages that happen to share a name, for example: "<b>math.Vector</b>" and "<b>java.util.Vector</b>"</p>