NMCT

NEW MEDIA &
COMMUNICATION
TECHNOLOGY

Introduction To Java

A quick start for C# developers

What’s up today?

- What can we learn from Hello World.
- How to compile and execute a basic java program.
- Learning more from “Hello World”.

> “*Inexperienced programmers think*
> *syntax is the biggest step.*
>
> *Experienced developers know*
> *syntax is the smallest step*”

```
package be.howest.java;

public class Main {
   public static void main(String[] args) {
       System.out.println("Hello World");
   }
}
```

You know what the code does, but could you write it yourself?

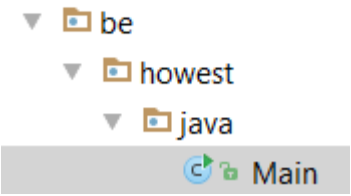
Lets see what we can learn from “hello world”

```java
package be.howest.java;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

```csharp
namespace Howest
{
    class Main
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello World");
        }
    }
}
```

```java
package be.howest.java;

public class Main {
   public static void main(String[] args) {
       System.out.println("Hello World");
   }
}
```

- package **keyword** & package **statement**
- A package describes **a folder structure**



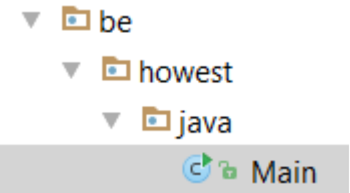
- Reverse FQDN als package naam
  `howest.be → be.howest`
  `google.com → com.google`
- Package names are **lowercase**
- The package statement should be the first statement in the file.

```java
package be.howest.java;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

```csharp
namespace Howest
{
    class Main
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello World");
        }
    }
}
```

```java
package be.howest.java;

public class Main {
   public static void main(String[] args) {
       System.out.println("Hello World");
   }
}
```

- Each file may contain **one public class.**
- and **multiple friendly classes**
- the **filename must match** the name of the public class



- Class names are **Upper Camel Case**, thus, filenames are also upper camel case. *(Even on windows!)*
- The extension of java files is **".java"**

Access Modifiers in Java

	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier (friendly)	Y	Y	N	N
private	Y	N	N	N

```java
package be.howest.java;

public class Main {
   public static void main(String[] args) {
       System.out.println("Hello World");
   }
}
```

```csharp
namespace Howest
{
   class Main
   {
       static void Main(string[] args)
       {
           System.Console.WriteLine("Hello World");
       }
   }
}
```

```java
package be.howest.java;

public class Main {
   public static void main(String[] args) {
       System.out.println("Hello World");
   }
}
```

- the **static** keyword can be used to indicate something is part of the class instead of the instance
- fields and methods, even static, are friendly by default.
- static can only be used on class members (fields, methods and inner classes)
- methods are **Lower Camel Case**
- **“main”** is the entry point for the Java Virtual Machine.

```java
package be.howest.java;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

```csharp
namespace Howest
{
    class Main
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello World");
        }
    }
}
```

```java
package be.howest.java;

public class Main {
   public static void main(String[] args) {
       System.out.println("Hello World");
   }
}
```

- In java, String is a reference type. (Notice the capital)

Primitive Types In Java

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

- Every **primitive (value) type** has a **wrapper (reference) type**

- Java uses **autoboxing** to convert from primitive to wrapper type. **(Beware converting null)**

```java
int iPrimitive = new Integer(0);
Integer iObject = 0;
```

- Java does not support User-Defined Value Types

Primitive Types In Java

```java
double d = 1;
float f = 2f;
d = f;
f = (float)d;

String s = "1";
int i = Integer.parseInt(s);
s = Integer.toString(i);
s = "" + 1;
```

- Conversion between types is almost the same as in C#.

```java
package be.howest.java;

public class Main {
   public static void main(String[] args) {
       System.out.println("Hello World");
   }
}
```

- System is a **class**, automatically imported from the java.lang package

```csharp
namespace Howest
{
   class Main
   {
       static void Main(string[] args)
       {
           System.Console.WriteLine("Hello World");
       }
   }
}
```

- System is a **namespace**.

```java
package be.howest.java;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

- out is a **“public static” field** of the System class
- out contains a reference to an instance of the Printstream class

```csharp
namespace Howest
{
    class Main
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello World");
        }
    }
}
```

- Console is a **static class** in the System namespace

```java
package be.howest.java;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

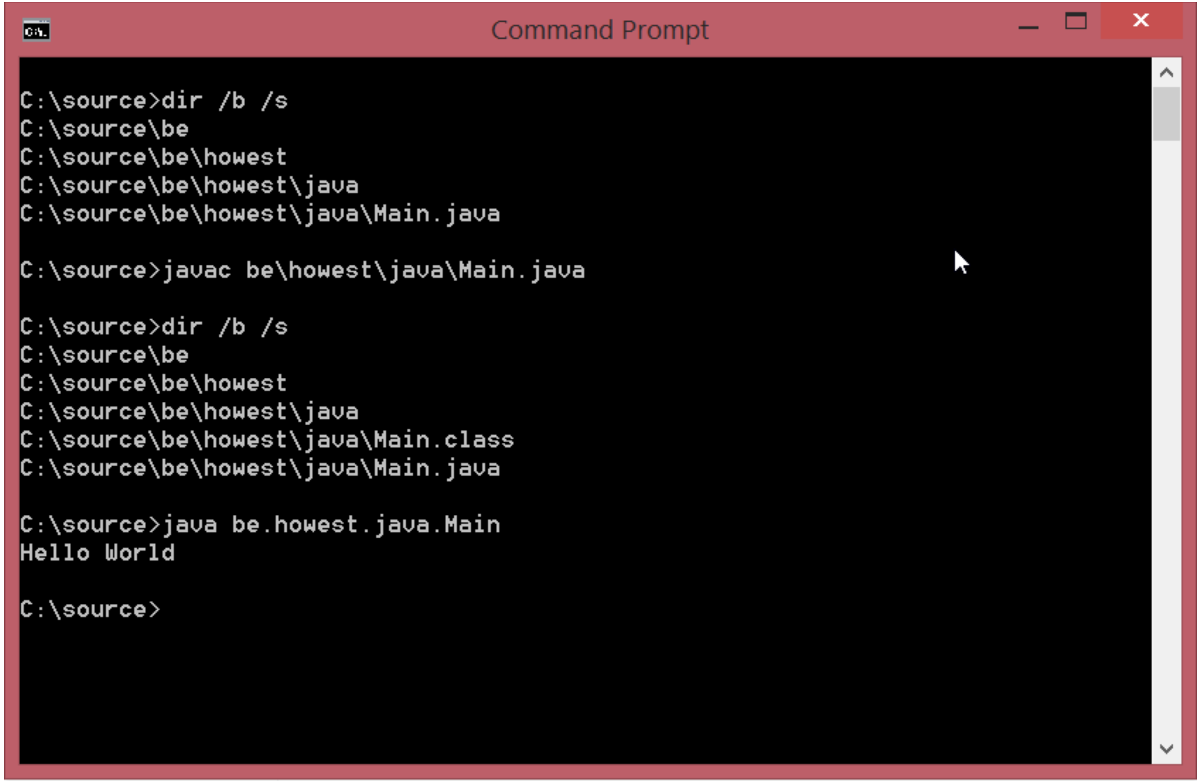
- println is an **instance method** of out, defined in the PrintSteam class

```csharp
namespace Howest
{
    class Main
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("Hello World");
        }
    }
}
```

- WriteLine is a **static method** of the Console class

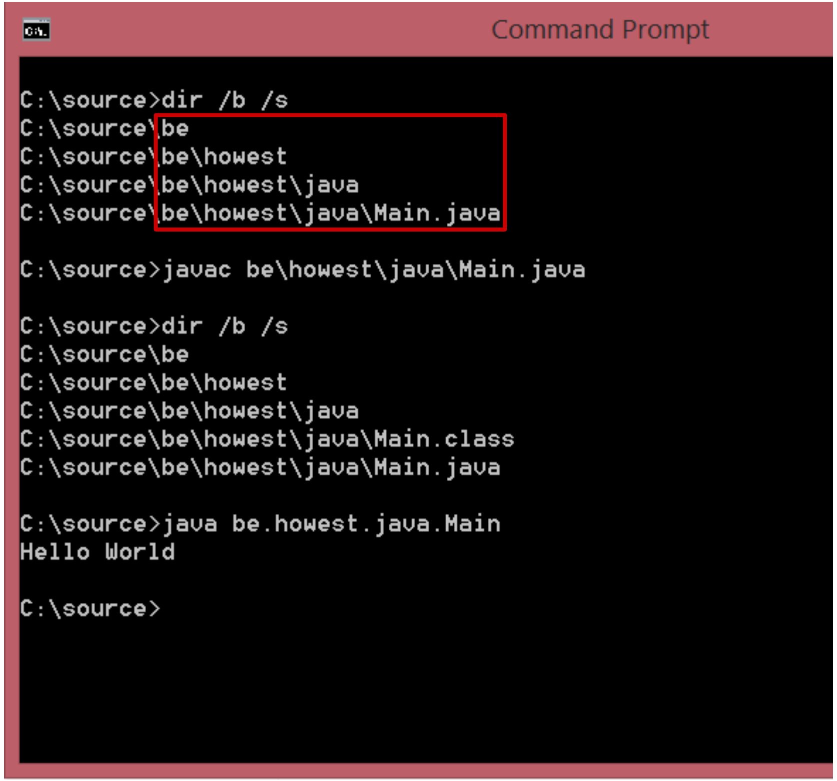
So, … Recapitulate!

Static, Class, Object & Instance, Method, Field, Access Modifier, Public, Private, Protected, Friendly, Main, Package, Source File, Primitive Type, Value Type, Reference Type, Argument, Import



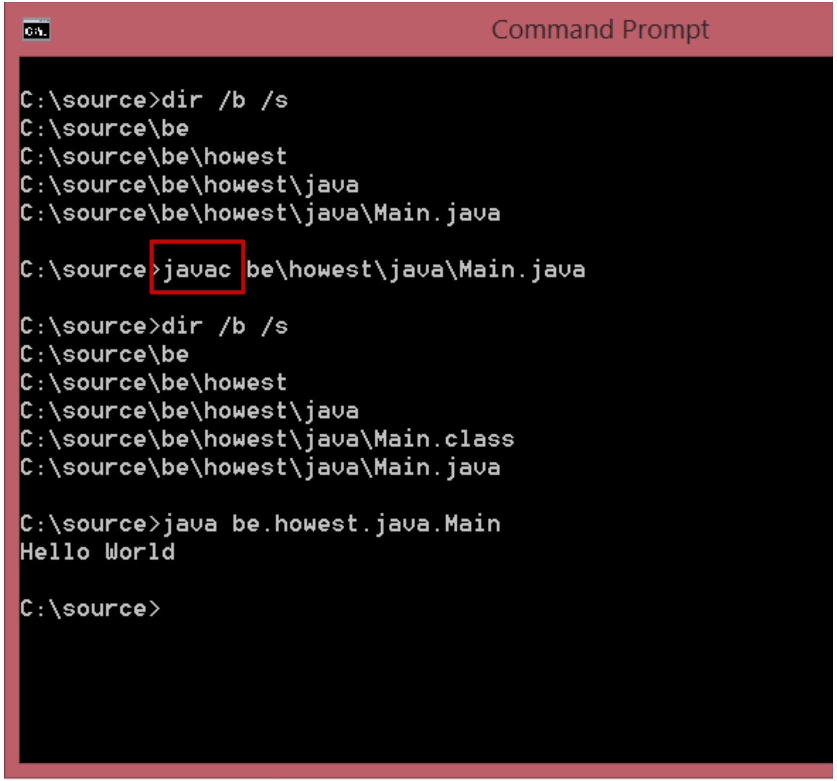
```
C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.java

C:\source>javac be\howest\java\Main.java

C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.class
C:\source\be\howest\java\Main.java

C:\source>java be.howest.java.Main
Hello World

C:\source>
```

How to compile from source to bytecode and how to execute bytecode



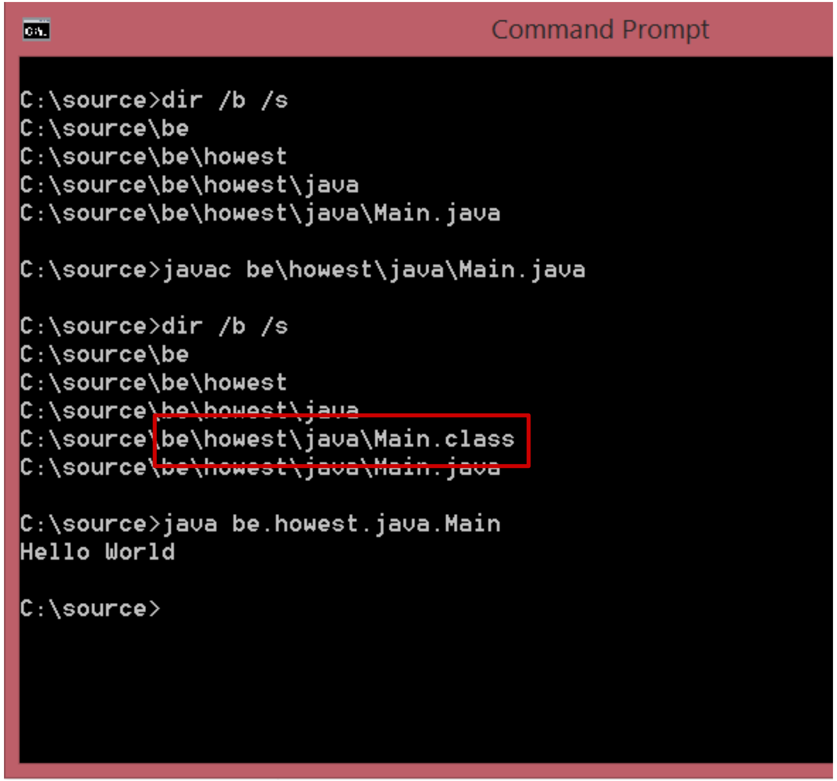
```
C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.java

C:\source>javac be\howest\java\Main.java

C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.class
C:\source\be\howest\java\Main.java

C:\source>java be.howest.java.Main
Hello World

C:\source>
```

- Our package structure and our Source file Main.java



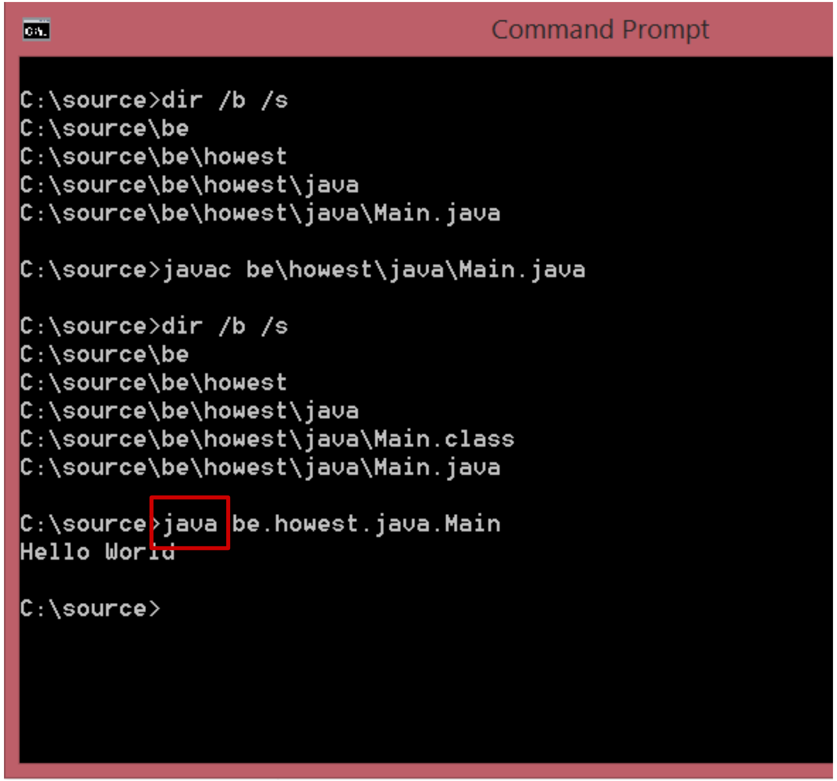
```
C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.java

C:\source>javac be\howest\java\Main.java

C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.class
C:\source\be\howest\java\Main.java

C:\source>java be.howest.java.Main
Hello World

C:\source>
```

- Our package structure and our Source file Main.java
- Using the “Java Compiler” javac to convert the source file to “Java Bytecode”



```
C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.java

C:\source>javac be\howest\java\Main.java

C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.class
C:\source\be\howest\java\Main.java

C:\source>java be.howest.java.Main
Hello World

C:\source>
```

- Our package structure and our Source file Main.java
- Using the “Java Compiler” javac to convert the source file to “Java Bytecode”
- Our Compiled Java Bytecode file a.k.a. Class File



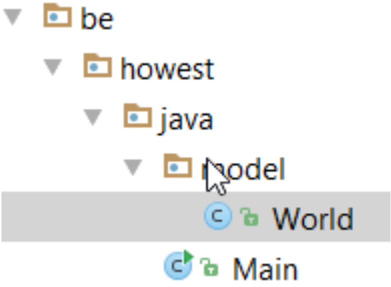
```
C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.java

C:\source>javac be\howest\java\Main.java

C:\source>dir /b /s
C:\source\be
C:\source\be\howest
C:\source\be\howest\java
C:\source\be\howest\java\Main.class
C:\source\be\howest\java\Main.java

C:\source>java be.howest.java.Main
Hello World

C:\source>
```

- Our package structure and our Source file Main.java
- Using the “Java Compiler” javac to convert the source file to “Java Bytecode”
- Our Compiled Java Bytecode file a.k.a. Class File
- Using the java command to instructing the java virtual machine to execute the be.howest.java.Main class.

```java
package be.howest.java.model;
/**
* Created by verborghs.
*/
public class World {
    private String greeting = "hello";
    private String name;

    public World(String name) {
        this.name = name;
        System.out.println(name + " created");
    }

    public World() {
        this(World.class.getSimpleName());
    }

    public void speak() {
        System.out.println(greeting + " " + name);
    }
}
```

```
package be.howest.java.model;
/**
* Created by verborghs.
*/
public class World {
    private String greeting = "hello";
    private String name;

    public World(String name) {
        this.name = name;
        System.out.println(name + " created");
    }

    public World() {
        this(World.class.getSimpleName());
    }

    public void speak() {
        System.out.println(greeting + " " + name);
    }
}
```

- Adding a new class, in another package



```java
package be.howest.java.model;
/**
* Created by verborghs.
*/
public class World {
   private String greeting = "hello";
   private String name;

   public World(String name) {
       this.name = name;
       System.out.println(name + " created");
   }

   public World() {
       this(World.class.getSimpleName());
   }

   public void speak() {
       System.out.println(greeting + " " + name);
   }
}
```

- Two kinds of comments:
  Multiline: `/*  ..*/`
  SingleLine `//`

- A special kind of comment:
  JavaDoc

```
/**
* Some Documentation
*/
```

```java
package be.howest.java.model;
/**
* Created by verborghs.
*/
public class World {
    private String greeting = "hello";
    private String name;

    public World(String name) {
        this.name = name;
        System.out.println(name + " created");
    }

    public World() {
        this(World.class.getSimpleName());
    }

    public void speak() {
        System.out.println(greeting + " " + name);
    }
}
```

- Class is called World and public …. So it is in a file World.java

```
package be.howest.java.model;
/**
* Created by verborghs.
*/
public class World {
    private String greeting = "hello";
    private String name;

    public World(String name) {
        this.name = name;
        System.out.println(name + " created");
    }

    public World() {
        this(World.class.getSimpleName());
    }

    public void speak() {
        System.out.println(greeting + " " + name);
    }
}
```

- this class has two private fields.
- We can access them with this.<fieldname> of if there is no ambiguity by just specifying the name.
- Fields are initialized to their ‘0’/’null’ values by default.
- The accessibility of fields is determined by their access modifiers.

```java
package be.howest.java.model;
/**
* Created by verborghs.
*/
public class World {
   private String greeting = "hello";
   private String name;

   public World(String name) {
       this.name = name;
       System.out.println(name + " created");
   }

   public World() {
       this(World.class.getSimpleName());
   }

   public void speak() {
       System.out.println(greeting + " " + name);
   }
}
```

- The constructor carries the same name as the Class.
- If we don’t supply our own constructor (s), we get a “default constructor”
- Constructors can be used to initialize fields
- after a constructor returns, an object instance is fully initialized. (Or: while the constructor is running, an object is not yet initialized)

```java
package be.howest.java.model;
/**
* Created by verborghs.
*/
public class World {
    private String greeting = "hello";
    private String name;

    public World(String name) {
        this.name = name;
        System.out.println(name + " created");
    }

    public World() {
        this(World.class.getSimpleName());
    }

    public void speak() {
        System.out.println(greeting + " " + name);
    }
}
```

- We can have multiple constructors. As long as their signature is different.
- The signature of a constructor is defined by his argument types. (Not their names)
- We can chain constructors by using **this()** as the first call to any other constructor in that class.

```java
package be.howest.java.model;
/**
* Created by verborghs.
*/
public class World {
   private String greeting = "hello";
   private String name;

   public World(String name) {
       this.name = name;
       System.out.println(name + " created");
   }

   public World() {
       this(World.class.getSimpleName());
   }

   public void speak() {
       System.out.println(greeting + " " + name);
   }
}
```

- We have a **method** speak.
- We can have multiple methods with the same name as long as the signature is different.
- The signature of a method is defined by its name, argument types and declared exceptions (Or: the return type is of no influence)
- The speak method has no return type and as such declares it as void.

```java
public class World {
   private String greeting = "hello";

  /*... code hidden for brevity ... */

   public String getGreeting() {
       return greeting;
   }

   public void setGreeting(String greeting) {
       this.greeting = greeting;
   }

   public String getSentence() {
       return greeting + " " + name;
   }

}
```

- Java does not have literals to define properties.
- this class has a R/W **property** greeting and a R/O property Sentence
- Do not confuse the property greeting with the **field** greeting
- A property is R/W when we have a **Getter** and a **Setter**
- A property is R/O when we only have a Getter.
- This is all just a **convention** defined by the Java Bean Specification

```java
package be.howest.java;

import be.howest.java.model.World;

public class Main {

   public static void main(String[] args) {
       World earth = new World();
       earth.speak();
   }

}
```

```java
package be.howest.java;

import be.howest.java.model.World;

public class Main {

   public static void main(String[] args) {
       World earth = new World();
       earth.speak();
   }

}
```

- The **import** keyword allows us to use class(es) which are in other packages.

- We can import **all classes** from a package:

  ```java
  import be.howest.java.model.*;
  ```

- We can import one **specific class** from a package

  ```java
  import be.howest.java.model.World;
  ```

- Or we can just not import a class and specify its **full class name** (which includes the package name).

- Using the full classname is useful when the there are multiple class with the same **simple class name**.

```java
public class Util {

   public final static int KM_PER_AU = 149597871;

   public static double auToKm(double au){
       return au * KM_PER_AU ;
   }

}
```

```
public class Util {

   public final static int KM_PER_AU = 149597871;

   public static double auToKm(double au){
       return au * KM_PER_AU ;
   }

}
```

- we can use the **static** modifier to indicate that a variable or method is part of the class instead of the object.

- Static members are accessible through the class, so we don’t need an instance to use them:

```
Util.auToKm(1.0);
```

```
public class Util {

    public final static int KM_PER_AU = 149597871;

    public static double auToKm(double au){
        return au * KM_PER_AU ;
    }

}
```

- we can use the **final** modifier to indicate that a value once initialized will not be changed.

- final variables are initialized after declaration, when the are not static the can be initialized in the constructor.

- **final local variables** are placed on the heap instead of the stack

```java
package be.howest.java.model;

import static be.howest.java.model.Util.auToKm;

public enum Planets {
    Mercury(4.0),Venus(0.7),Earth(1.0),
    Mars(1.5), Jupiter(5.2),Saturn(9.5),
    Uranus(19.2),Neptune(30.1);

    private double distanceFromSun;

    Planets(double distanceFromSun) {
        this.distanceFromSun = distanceFromSun;
    }

    public double getDistanceFromSun() {
        return auToKm(distanceFromSun);
    }

}
```

```java
package be.howest.java.model;

import static be.howest.java.model.Util.auToKm;

public enum Planets {
   Mercury(4.0),Venus(0.7),Earth(1.0),
   Mars(1.5), Jupiter(5.2),Saturn(9.5),
   Uranus(19.2),Neptune(30.1);

   private double distanceFromSun;

   Planets(double distanceFromSun) {
       this.distanceFromSun = distanceFromSun;
   }

   public double getDistanceFromSun() {
       return auToKm(distanceFromSun);
   }

}
```

- we can use **import static** to import static members of a class. This allows us to use a shorthand for those imported methods.

- We can either import everything using ‘*’ or one member by specifying the name of said member. (Just like normal imports)

```java
package be.howest.java.model;

import static be.howest.java.model.Util.auToKm;

public enum Planets {
   Mercury(),Venus(),Earth(),
   Mars(), Jupiter(),Saturn(),
   Uranus(),Neptune();
}
```

- The first line of an enum should be the list of possible enumerations.

```java
package be.howest.java.model;

import static be.howest.java.model.Util.auToKm;

public enum Planets {
   Mercury(),Venus(),Earth(),
   Mars(), Jupiter(),Saturn(),
   Uranus(),Neptune();

   private double distanceFromSun;

   public double getDistanceFromSun() {
       return auToKm(distanceFromSun);
   }

}
```

- Just like classes enums may have methods and fields.

```java
package be.howest.java.model;

import static be.howest.java.model.Util.auToKm;

public enum Planets {
   Mercury(4.0),Venus(0.7),Earth(1.0),
   Mars(1.5), Jupiter(5.2),Saturn(9.5),
   Uranus(19.2),Neptune(30.1);

   private double distanceFromSun;

   Planets(double distanceFromSun) {
       this.distanceFromSun = distanceFromSun;
   }

   public double getDistanceFromSun() {
       return auToKm(distanceFromSun);
   }

}
```

- Enums are allowed to have a friendly constructor.
- In actuality, an enum is nothing more than syntactical sugar for a class.
- And as we will see later on, they can be used in a switch-case control structure.

```java
package be.howest.java;

import be.howest.java.model.Planets;

public class Main {

    public static void main(String[] args) {
        Planets[] planets = Planets.values();

        for(int i = 0; i < planets.length; i++) {
            System.out.println( planets[i].name() );
        }

    }

}
```

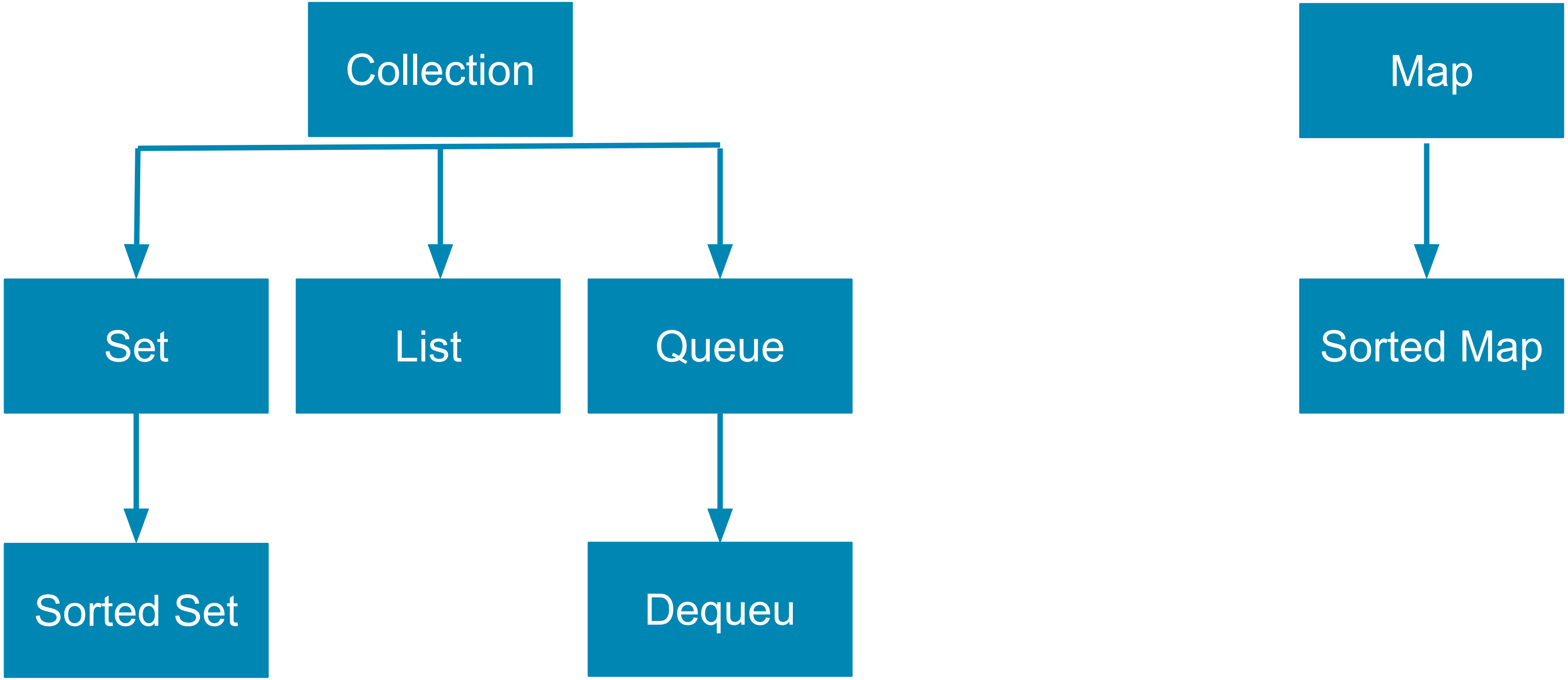
- Java of course supports the classical way of looping using **for-loop** control structures.

```java
package be.howest.java;

import be.howest.java.model.Planets;

public class Main {

   public static void main(String[] args) {
       Planets[] planets = Planets.values();

       for(Planets planet : Planets.values()) {
           System.out.println( planet.name() );
       }

   }

}
```

- If the index of the elements is not required we can use the **enhanced for-loop** control structure.

```java
package be.howest.java;

import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import be.howest.java.model.Planets;

public class Main {

   public static void main(String[] args) {

       List<Planets> planets =
               Arrays.asList(Planets.values());

       Iterator<Planets> iter = planets.iterator();

       while(iter.hasNext()) {
           Planets planet = iter.next();
           System.out.println(planet.name());
       }
   }
}
```

- Most modern languages allow us to use **List**s instead of **Array**s
- In java all Standard **Collection**s are part of the **java.util** package


Collection
Set
List
Queue
Sorted Set
Dequeu
Map
Sorted Map

	Hash	Array	Tree	Linked	Hash + Linked
Set	**HashSet**		TreeSet		LinkedHashSet
List		**ArrayList**		LinkedList	
Queue				LinkedList	
Deque		**ArrayDeque**		LinkedList	
Map	**HashMap**		TreeMap		LinkedHashMap

```java
package be.howest.java;

import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import be.howest.java.model.Planets;

public class Main {

   public static void main(String[] args) {

       List<Planets> planets =
               Arrays.asList(Planets.values());

       Iterator<Planets> iter = planets.iterator();

       while(iter.hasNext()) {
           Planets planet = iter.next();
           System.out.println(planet.name());
       }
   }
}
```

- To allow us to modify and use arrays easily, we can use **the Arrays class**.
- The most useful method probably being **asList()**
- If you do want to use the low level array, you might also find System.arrayCopy() or the other methods of the Arrays class useful.

```java
package be.howest.java;

import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import be.howest.java.model.Planets;

public class Main {

    public static void main(String[] args) {

        List<Planets> planets =
                Arrays.asList(Planets.values());

        Iterator<Planets> iter = planets.iterator();

        while(iter.hasNext()) {
            Planets planet = iter.next();
            System.out.println(planet.name());
        }
    }
}
```

- Before there was an enhanced for-loop, programmers had to use **iterators/enumerations** to loop over a collection.
- An iterator/enumeration is an object that allows one to go over each element of a **container.**
- Even though we have enhanced for-loops now, the iterator way of doing things returns in many libraries in one or another form.

```java
package be.howest.java;

import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import be.howest.java.model.Planets;

public class Main {

   public static void main(String[] args) {

       List<Planets> planets =
               Arrays.asList(Planets.values());

       Iterator<Planets> iter = planets.iterator();

       while(iter.hasNext()) {
           Planets planet = iter.next();
           System.out.println(planet.name());
       }
   }
}
```

- Collections, as many other classes, are **generic classes**.
- They are a construct on top of the old implementation, and as such they forget about their type once compiled.
- **Generic type parameters** will be used throughout many libraries.

```java
package be.howest.java;

import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import be.howest.java.model.Planets;

public class Main {

   public static void main(String[] args) {

       List<Planets> planets =
               Arrays.asList(Planets.values());

       Iterator<Planets> iter = planets.iterator();

       while(iter.hasNext()) {
           Planets planet = iter.next();
           System.out.println(planet.name());
       }
   }
}
```

- There is also are **while-loop** and **do-while-loop** control structures.

```java
package be.howest.java;

import be.howest.java.model.Planets;
import static be.howest.java.model.Util.KM_PER_AU;

public class Main {
    public static void main(String[] args) {
        for(Planets p: Planets.values()) {
            System.out.print(planet.name());

            if(p.getDistanceFromSun() > KM_PER_AU) {
                System.out.print(" is less than ");
            } else if(p.getDistanceFromSun() > KM_PER_AU) {
                System.out.print(" is more than ");
            } else {
                System.out.print(" is exactly ");
            }

            System.out.print("1 AU from the Sun");
        }
    }
}
```

- There are **if-else if-else** control structures.

```java
package be.howest.java;

import be.howest.java.model.Planets;

public class Main {
   public static void main(String[] args) {
       for(Planets planet : Planets.values()) {
          System.out.print(planet.name());

          switch (planet) {
              case Jupiter:
              case Neptune:
                  System.out.println(" has a 'p' ");
                  break;
              default:
                  System.out.println(" has no 'p' ");
          }

          System.out.println(" in its name");
       }
   }
}
```

- There is a **switch-case** control structure.
- It can handle **primitive types and enums** in it’s cases.
- Cases are fall-through, don’t forget to use **break**.
- We can have 1 **default** case.

```
public class Main {
   public static void main(String[] args) {
       List<String> fruits = new ArrayList<>();
       Collections.addAll(fruits, "cherry", "date", "apple", "banana");

       for(int i = 1; i < fruits.size(); i++) {
          String fruit = fruits.get(i);
          int j = i;
          while( j > 0) {
              if( fruit.compareTo(fruits.get(j-1)) > 0) {
                   break;
              } else {
                   fruits.set(j, fruits.get(j-1));
                   j--;
               }
          }
          fruits.set(j, fruit);
       }

       System.out.print(fruits);
   }
}
```

The package declaration and the import have been left out for brevity

```java
public class Main {
   public static void main(String[] args) {
       List<String> fruits = new ArrayList<>();
       Collections.addAll(fruits, "cherry", "date",
       "apple", "banana");

       for(int i = 1; i < fruits.size(); i++) {
          String fruit = fruits.get(i);
          int j = i;
          while( j > 0) {
              if( fruit.compareTo(fruits.get(j-1)) > 0) {
                   break;
              } else {
                   fruits.set(j, fruits.get(j-1));
                   j--;
               }
          }
          fruits.set(j, fruit);
       }

       System.out.print(fruits);
   }
}
```

- Using these control structures we can write a insertion sort, the simplest, but definitely not the best, sorting algorithm.
- We could of course learn about some new features to make sorting a list easier.
- Notice how we are using the **java.util. Collections** class to make our life easier.
- We could have used the add method of the collection to add each element individually.

```java
public class Main {
  public static void main(String[] args) {
     List<String> fruits = new ArrayList<>();
     Collections.addAll(fruits, "cherry", "date",
     "apple", "banana");

     for(int i = 1; i < fruits.size(); i++) {
       String fruit = fruits.get(i);
       int j = i;
       while( j > 0) {
         if( fruit.compareTo(fruits.get(j-1)) > 0) {
             break;
         } else {
             fruits.set(j, fruits.get(j-1));
             j--;
          }
       }
       fruits.set(j, fruit);
     }

     System.out.print(fruits);
  }
}
```

- Knowing your algorithms is a must!
- But if you know your languages, frameworks and libraries you don’t have to reimplement them.

```java
package be.howest.java;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main {
   public static void main(String[] args) {
       List<String> fruits = new ArrayList<>();
       Collections.addAll(fruits, "cherry", "date",
             "apple", "banana");

       Collections.sort(fruits);

       System.out.print(fruits);

   }

}
```

- Notice how we are using the **java.util.Collections** class to make our life easier.
- Have you heard about the internet? How about Google or Stackoverflow?
- Don’t Copy-n-Paste! Read, Learn, Do!

```java
package java.util;

public interface Comparator<T> {

   int compare(T o1, T o2);

   boolean equals(Object obj);

}
```

- `Comparator<T>` is an interface we can use to implements a custom way to compare objects of a generic type T
- This one is part of the Java Library, but we could define our own interfaces.

```
package java.util;

public interface Comparator<T> {

   int compare(T o1, T o2);

   boolean equals(Object obj);

}
```

- Interfaces and classes can be generic.

```java
package java.util;

public interface Comparator<T> {

   int compare(T o1, T o2);

   boolean equals(Object obj);

}
```

- They specify methods that should be defined in class that **implement** this interface.

```java
package be.howest.java.model;

import java.util.Comparator;

public class LengthComparator implements Comparator<String> {

   @Override
   public int compare(String s1, String s2) {
       return  s2.length() - s1.length();
   }

}
```

- Here we implement the interface for Strings.

```java
package be.howest.java.model;

import java.util.Comparator;

public class LengthComparator implements Comparator<String> {

   @Override
   public int compare(String s1, String s2) {
       return  s2.length() - s1.length();
   }

}
```

- The @override **annotation** is optional, but allows the compile to verify we didn't make any mistakes.

- Annotation can influence compile and runtime behaviour. They add useful **metadata**.

```java
package be.howest.java;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import be.howest.java.model.LengthComparator;

public class Main {

   public static void main(String[] args) {
       List<String> fruits = new ArrayList<>();
       Collections.addAll(fruits, "cherry", "date",
          "apple", "banana");

       Collections.sort(fruits, new LengthComparator());

       System.out.print(fruits);

   }
}
```

- We can now use this class to modify the behaviour of sort by specifying how it should compare the items in the List.

```java
package be.howest.java;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Main {
   public static void main(String[] args) {
       List<String> fruits = new ArrayList<>();
       Collections.addAll(fruits, "cherry", "date");
       Collections.sort(fruits, new Main.LengthComparator());
       System.out.print(fruits);
   }

   private static class LengthComparator
                 implements Comparator<String> {
       @Override
       public int compare(String s1, String s2) {
           return  s2.length() - s1.length();
       }
   }
}
```

- Sometimes it is useful to make the make the class an **inner class**.
- Here we use a **static inner class**.
- The same rules from static members apply to referencing a static inner.
- Static classes do not have access to the outer class.

```java
package be.howest.java.model;

public class World {
   private final Greeter greeter;

   private String name;
   public World(String name) {
       this.name = name;
       this.greeter = new Greeter();
   }

   public void speak() {
       greeter.sayHello();
   }

   private class Greeter {
       private String name = "Mark";
       public void sayHello() {
           System.out.println(name + " says: hello "
                 + World.this.name);
       }
   }
}
```

- We also have **non-static inner classes**
- This is a useful feature we will be using to keep our code clean or to export parts of a class as a different interface.

```
package be.howest.java.model;

public class World {
   private final Greeter greeter;

   private String name;
   public World(String name) {
       this.name = name;
       this.greeter = new Greeter();
   }

   public void speak() {
       greeter.sayHello();
   }

   private class Greeter {
       private String name = "Mark";
       public void sayHello() {
           System.out.println(name + " says: hello "
                 + World.this.name);
       }
   }
}
```

- these inner classes have access to members of the enclosing class.

- When there is a possible conflict, you need to use **<Class>.this** instead of just this.

```java
package be.howest.android.helloworld;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import static android.view.View.OnClickListener;

public class MainActivity extends Activity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       View view = findViewById(android.R.id.content);
       view.setOnClickListener(new ClickHandler());
   }

   class ClickHandler implements  OnClickListener {
       @Override
       public void onClick(View v) {
           Log.i("MainActivity", "Clicked");
       }
   }
}
```

- The inner classes are especially useful when we need to assign event handlers.

- When they are non-static, they can use the fields and members of the enclosing class.

```java
package be.howest.android.helloworld;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import static android.view.View.OnClickListener;

public class MainActivity extends Activity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       View view = findViewById(android.R.id.content);
       view.setOnClickListener(new OnClickListener() {
           @Override
           public void onClick(View v) {
               Log.i("MainActivity", "Clicked");
           }
       });
   }
}
```

- There are also anonymous inner classes.
- They allow us to implement an interface "in place".

```java
package be.howest.android.helloworld;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import static android.view.View.OnClickListener;

public class MainActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       View view = findViewById(android.R.id.content);
       view.setOnClickListener(new OnClickListener() {
           @Override
           public void onClick(View v) {
             onViewClicked(v);
           }
       });
   }
   private void onViewClicked(View v) {
       Log.i("MainActivity", "Clicked");
   }
}
```

- They are a verbose equivalent of delegates and event handlers in C#

```java
package be.howest.android.helloworld;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import static android.view.View.OnClickListener;

public class MainActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       View view = findViewById(android.R.id.content);
       view.setOnClickListener(new OnClickListener() {
           @Override
           public void onClick(View v) {
             onViewClicked(v);
           }
       });
   }
   private void onViewClicked(View v) {
       Log.i("MainActivity", "Clicked");
   }
}
```

- Inheritance is done using the **extends** keyword.
- There is only **Single Class Inheritance**.
- If you need more, you need to use interfaces.

```java
class Employee
extends Person
implements Retrievable, Persistable {}
```

```java
package be.howest.android.helloworld;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import static android.view.View.OnClickListener;

public class MainActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       View view = findViewById(android.R.id.content);
       view.setOnClickListener(new OnClickListener() {
           @Override
           public void onClick(View v) {
             onViewClicked(v);
           }
       });
   }
   private void onViewClicked(View v) {
       Log.i("MainActivity", "Clicked");
   }
}
```

- The **super** literal can be used to call methods of the parent class.
- We can use super just like this to call the constructor of the parent class.
- Methods that are accessible are always overridable unless we specify them as final using the **final** keyword.
- Classes can also be final.

```java
File f = new File("hello.txt");
try {
    f.createNewFile();
} catch (IOException e) {
    e.printStackTrace();
}
```

- One final thing: **Checked Exceptions**