Programming in Java

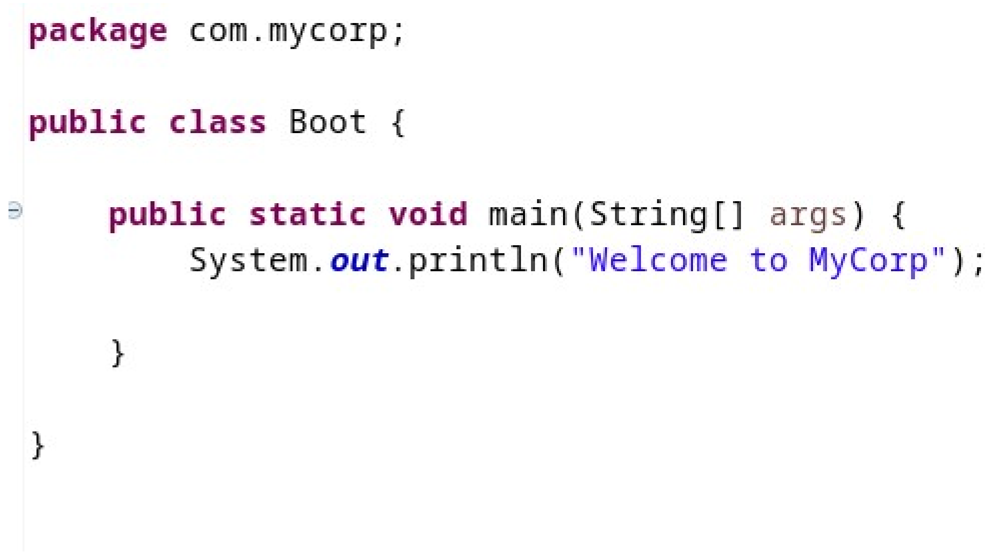
5. Packages

Java Packages

- Java packages combine two concepts
  - A directory like structure for organizing Java classes
  - Namespaces to avoid naming conflicts between classes in different package
  - Think of a namespace as being like an area code for phone numbers
- Packages as structure
  - A package can be thought of as a directory in a file system
  - It has a qualified name like a file system path name
  - It can contain various Java constructs like class definitions
  - But it can also contain other packages which enables a recursive organization
- Package are implemented as directories
  - Java is intended to be portable across file systems
  - Packages are an abstraction that is implemented into the local files system by the JRE
  - Dots are used as a directory structure
    - com.accounting.payroll → com/accounting/payroll   or com\accounting\payroll

The package Statement

- There may be many files in a single package
  - There is no analogue to a directory table in a package
  - This is very OS dependent
  - Trying to have the package index or keep track of its contents is not technically feasible
- Instead, each file remembers which package it should be in
  - This is the package statement which is the first line in every file
  - There is a default unnamed package where files without package declarations go
  - This is only used for small quick and dirty code
  - Using it consistently is considered poor Java style



```
package com.mycorp;

public class Boot {

    public static void main(String[] args) {
        System.out.println("Welcome to MyCorp");

    }

}
```

The Bootstrap Problem

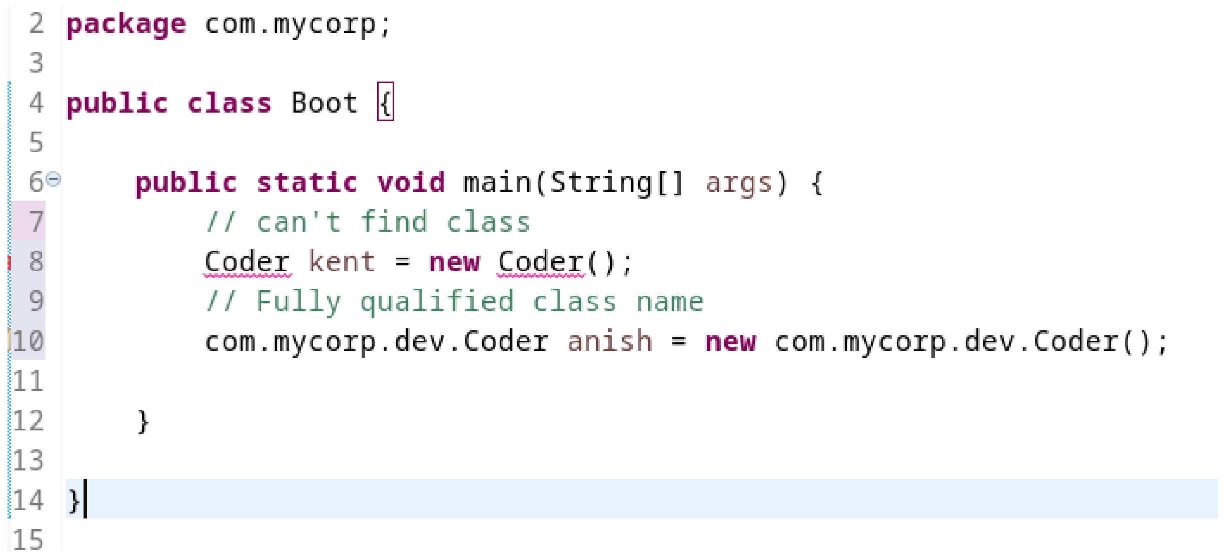
- All programs have to start somewhere, often called the bootstrap code
  - Most compiled programming languages have a main() function that represents the entry point to the program execution
- The problem is that Java only allows methods (functions) inside classes
  - That means that we have to stick a main() method is some class
  - Java doesn’t care where it is, as long as it can find it
- This is problematic for several reasons
  - Putting the main() method in an arbitrary location makes it hard to find
  - Whatever class it is in now has an additional responsibility – starting the application
- Best practice
  - Create a special class with only one responsibility – to run the main() method
  - And put the class in a specific location, the topmost package for example, so that it does not pollute the rest of the code

Package as Namespace

- For this discussion we are going to only be referring to class definitions
  - This is just to illustrate the concept of visibility
  - We will refine this later when we work with classes in more detail
- Java uses *package* visibility by default
  - This means that every class in a package can refer to every other class in the same package
- Some classes can be declared to be *public*
  - This means that classes outside the package can also refer to it
- Java has a couple of special rules for public classes
  - There can only be one public class defined per file
  - The file has to have the same name as the public class
  - This makes it easier for Java to manage loading the classes into the JVM at startup

Fully Qualified Class Names

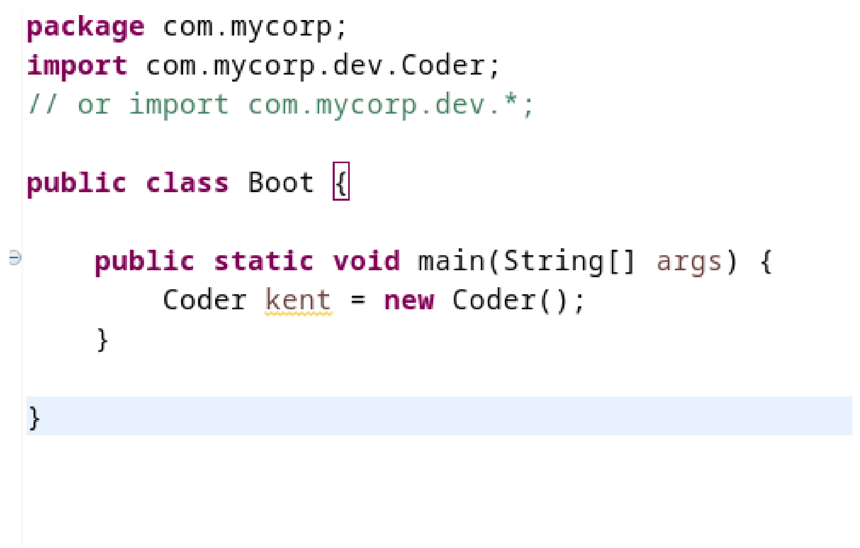
- When code in one package refers to a public class in another package
  - Java has to find that definition.. somewhere
  - There is no index so just using the class name alone is pointless
- One way to help Java out is to prefix the class name with a full qualified name which is done by adding the package name as a prefix to the class name
  - This is like using a fully qualified path name in a file system



```java
package com.mycorp;

public class Boot {

    public static void main(String[] args) {
        // can't find class
        Coder kent = new Coder();
        // Fully qualified class name
        com.mycorp.dev.Coder anish = new com.mycorp.dev.Coder();

    }

}

```

The import Statement

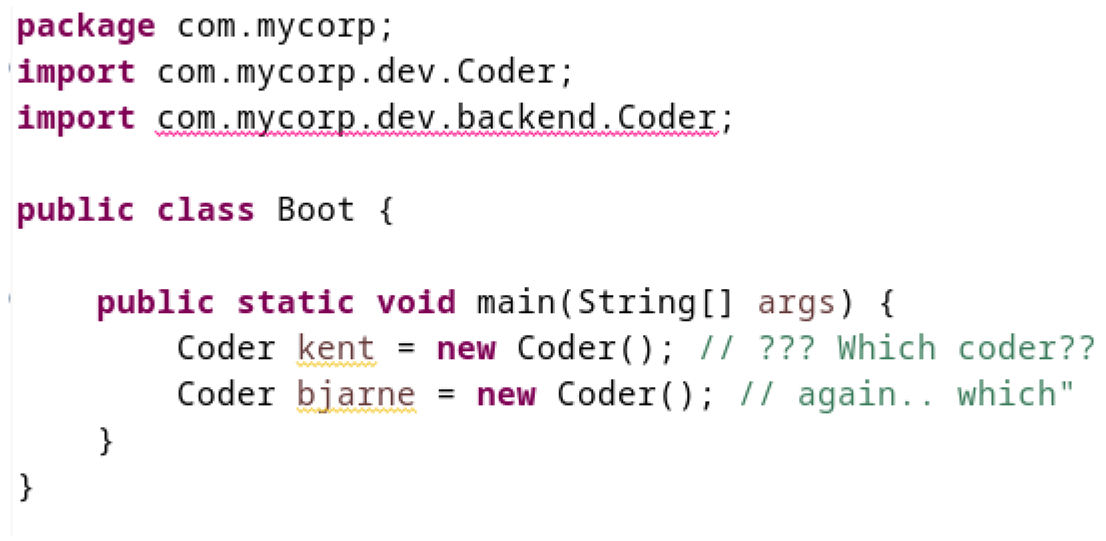
- Since it’s a pain to use full qualified names, using the import statement makes easier
  - The full qualified class name is placed after the package statement in an import statement
  - This directs Java where to look to find the definition of the imported class
  - The import statement doesn’t move anything, just allows Java to use the class name as a alias for the imported fully qualified name
- If there are a number of classes to be imported from a package the wildcard can be used instead of listing all the classes to be imported.



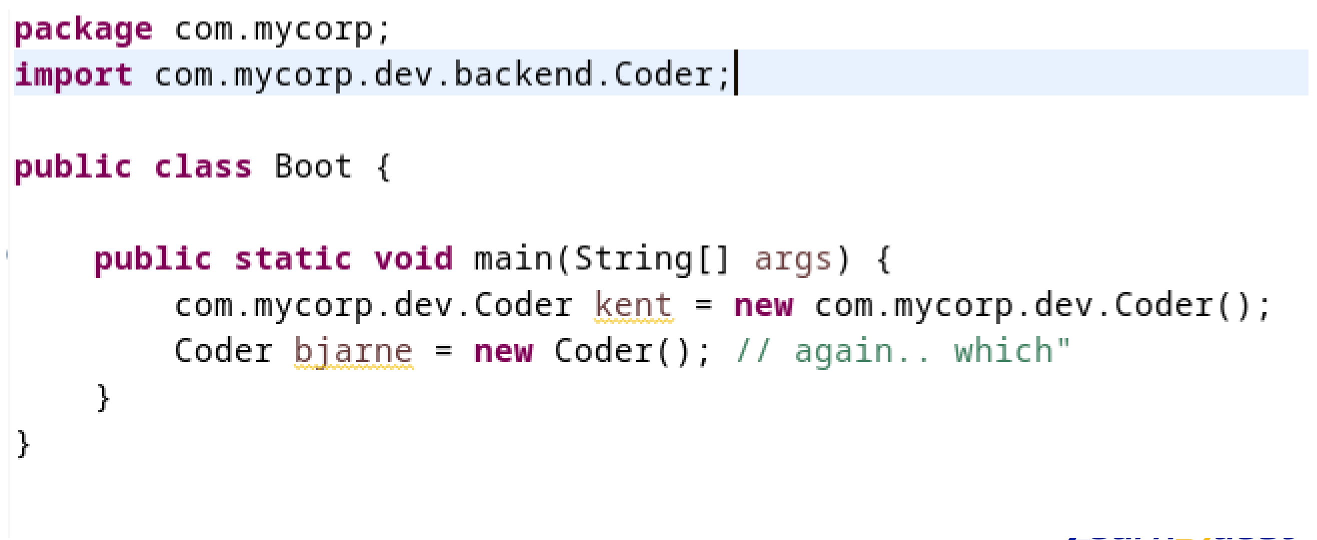
```java
package com.mycorp;
import com.mycorp.dev.Coder;
// or import com.mycorp.dev.*;

public class Boot {

    public static void main(String[] args) {
        Coder kent = new Coder();
    }

}
```

Naming Conflicts

- It may may happen that an imported class name may conflict with another class
  - There may already be a class with that name in the package
  - There maybe two different classes with the same name being imported from two different packages
  - This is a name space collision
- In this case, to avoid ambiguity, one of the imported classes will have to use its full qualified name
- Notice in the example, the first import had to be removed
  - Java is only concerned with import statements when looking for collisions



```
package com.mycorp;
import com.mycorp.dev.Coder;
import com.mycorp.dev.backend.Coder;

public class Boot {

    public static void main(String[] args) {
        Coder kent = new Coder(); // ??? Which coder??
        Coder bjarne = new Coder(); // again.. which"
    }
}
```



```
package com.mycorp;
import com.mycorp.dev.backend.Coder;

public class Boot {

    public static void main(String[] args) {
        com.mycorp.dev.Coder kent = new com.mycorp.dev.Coder();
        Coder bjarne = new Coder(); // again.. which"
    }
}
```

Demo

Working with Java Packages

Lab 5-1

Java Packages

Java™