Packages, Imports and Naming Conflicts

In this module we'll be going over packages.
Programmers all over the world are writing Java code and classes, in order to share and use those classes Java needs a way to organize them. This problem is solved with packages!

Packages are used in java to group related code in a `namespace`. A `namespace` is a technical term for a the identifier programmers give this grouped code that give Java the information it needs to find it

Questions:

1. Why would it be important to group code together?
2. Why do packages need names?

Using packages

How will the following code execute?

```
public class RandomWithoutImport {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(10))
  }
}
```

What about this code?

```
import java.utli.Random; //The package!!!

public class RandomWithoutImport {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(10))
  }
}
```

What about this code?

```
import java.util.Random; //The package!!!

public class RandomWithoutImport {
  public static void main(String[] args) {
    Random r = new Radnom();
    System.out.println(r.nextInt(10))
  }
}
```

What about this code?

```
import java.util.Random; //The package!!!

public class RandomWithoutImport {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(10))
  }
}
```

The important takeaway is that packages give java the information that it needs to use code either you or others have written! Because of this it is important to group your code logically with packages and then explicitly import the classes you want from the correct package. We're going to learn how to do this today.

Imagine you are cooking a meal at your house with a friend. You might ask your pal to grab an ingredient for you. If so you would probably say something like:

> Would you please grab me the cinnamon from the top shelf in the pantry

In this case we start with the most specific request: "cinnamon", then a more generalized instruction "top shelf" and finally the most general "pantry"
Just like we give our friend specific instructions about how to find what they are looking for, we must also give Java specific instructions.

Diifferences are that in Java we cut out a lot of the unnecessary words, use periods to separate instructions, and go from most general to most specific.
So if there was an imaginary package for a cinnamon class and we wanted to tell java to use it the instructions would be:

Redundant Imports

Although imports are very specific, they involve typing. And typing is something that many programmers seem to always be trying to avoid. Ironic, isn't it? But true. To satisfy these programmers who want to type as little, Java has a convenience: `redundant imports`.

An example of this is the `java.lang` package.
`java.lang` is automatically imported, and lets us use all of its classes without explicitly importing them. This package contains many useful tools, but the one you are probably most familiar with is `System.out.println()`.

Another example of this is packages that live in the same package as the code being executed. Java assumes that you would like to have all of the classes available to you when you are working within a package.

Static Imports

Classes can have a special type of member that is identified with the keyword `static`. When a new object is instantiated, typically its members (properties, methods) are instantiated with the object. Static members are different - they are instantiated not with the object, but with the class. This means that even if you have a million objects instantiated from a class, there will only be one instance of its static members.

```java
//when the program starts, all static members of classes a
package com.ryanmoore.staticExample;
public class StaticAndNotStatic {
  public int notStaticMember 1;
  public static int staticMember = 50;

  public StaticAndNotStatic() {
    System.out.println(this.notStaticMember)
    System.out.println(this.staticMember);
  }
}

import com.ryanmoore.staticExample.staticAndNotStatic
public class Main {
  public static void main(String[] args) {
    //each time an object is instantiated all non static m
    StaticAndNotStatic a = new StaticAndNotStatic();
    StaticAndNotStatic b = new StaticAndNotStatic();
  }
}
```

Since static members are created when the program first runs and they live on the class level, that means we can use them without instantiating an object.
Java allows you to import and use static members directly with the syntax `import static packageName.className.staticMemberName`

So if we wanted to use the static member of the earlier example:

```
import static com.ryanmoore.staticExample.staticAndNotSta
public class Main {
  public static void main(String[] args) {
    System.out.println(staticMember);
  }
}
```

Wildcard Imports

The `*` character can be used to import all the classes in a package, it is a short hand convenience that allows us to use a lot of classes without having to go line by line importing each one:

```
import pantry.topShelf.*
```

Naming Conflicts

Sometimes two or more classes will be have the same thing. In order to tell Java which one you intend to use, you must use a fully qualified name instead of importing each. When you use the fully qualified name you do not have to use the import syntax.

```
public class WildcardImports {
  public static void main(String[] args) {
    Pepper p = new Pepper();
    //since the package and class names are specified java
    pantry.topShelf.Cinnamon c1 = new pantry.topShelf.Cinn
    pantry.bottomShelf.Cinnamon c2 = new pantry.bottomShe
  }
}
```

Creating Packages

There are a few rules and guidelines to keep in mind when creating packages:

1. You cannot use reserved keywords to name packages
2. Packages should match directory structure
3. Packages should be grouped logically

Another common and best practice for creating package names is to use the reverse domain naming scheme.

This means you start with `com`, follow with your company or your name, follow with the project name, and then the package name.

For example, I work for MX, so if I was developing a battleship game for MX I could organize my code like this:

directory:
com/
com/mx/
com/mx/battleship/
com/mx/battleship/ships
com/mx/battleship/board
com/mx/batleship/players

package structure
com.mx.battleship
com.mx.battleship.ships
com.mx.battleship.board
com.mx.battleship.players

This can be a little confusing, because we use a package naming convention based on web addresses even when we aren't developing web applications, but it ensures that even if many, many people write battleship applications, we can share our code without unexpected naming conflicts.

The Default Package

Any time you write code without specifying a package it actually goes into the `Default Package`. This is what we've been doing for most of this class so far. The default package is convenient for practicing, but should be avoided for production code because you don't get the benefits of grouping your code logically or the guarantees packages provide that Java will find the classes you intended to use!

Order of Elements in a Class

Element	Example	Required?	Order
Package declaration	package com.mx.battleship	No	First line .
Import statements	import com.mx.battleship.Game	No	Immediately after package
Class declaration	public class Game	Yes	Immediately after the import
Field declarations	int numPlayers;	No	Anywhere in class
Method declarations	void startGame()	No	Anywhere in class

```
package com.mx.battleship; //package first

import com.mx.battleship.Game.Player; //then import

public class Main() { //then class declaration - Required
  public numPlayers; //field declaration anywhere in clas
  public void startGame() { //method declaration anywhere
    System.out.println("playing game");
  }
}
```

Additional Resources

- Working with Packages by Jim Wilson
  https://app.pluralsight.com/player?course=java-fundamentals-language&author=jim-wilson&name=java-fundamentals-language-m11&clip=0&mode=live
- Java Docs: Creating and Using Packages
  https://docs.oracle.com/javase/tutorial/java/package/packages.html
- Oracle Tutorial: Packages
  https://docs.oracle.com/javase/tutorial/java/package/index.html