ALGONQUIN COLLEGE

MODULE 02: JAVA FOUNDATIONS

Professor : Dave Houtman
Office: T323
Office Hrs: Wednesday 14:15 – 15:30
Wednesday (after lecture*)
* confirm beforehand

Email: houtmad@algonquincollege.com

2.0 'Hello World'

'Hello World' represents the simplest possible program that can be executed in any language. Hence it often forms the starting point for many programming books and courses.

A sample in Java is shown here:

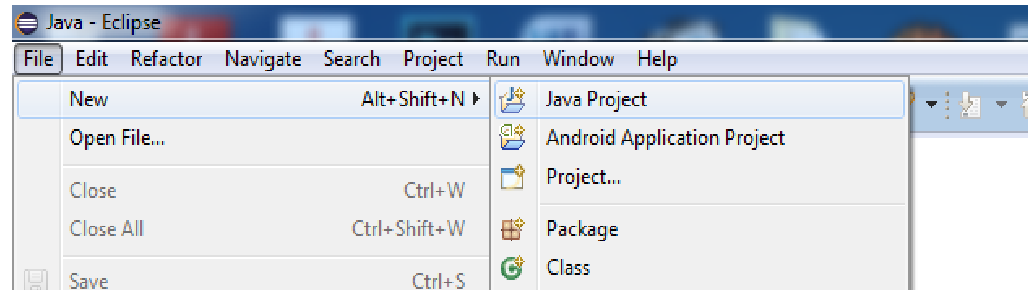
```
public class HelloWorld {
     public static void main(String[] args) {
      System.out.println ("Hello World");
     }
}
```

In Java, as in most programming languages, things are often not so simple as they first appear. There's a great deal going on here. Let's look at this program in much greater detail.

2.1 Packages

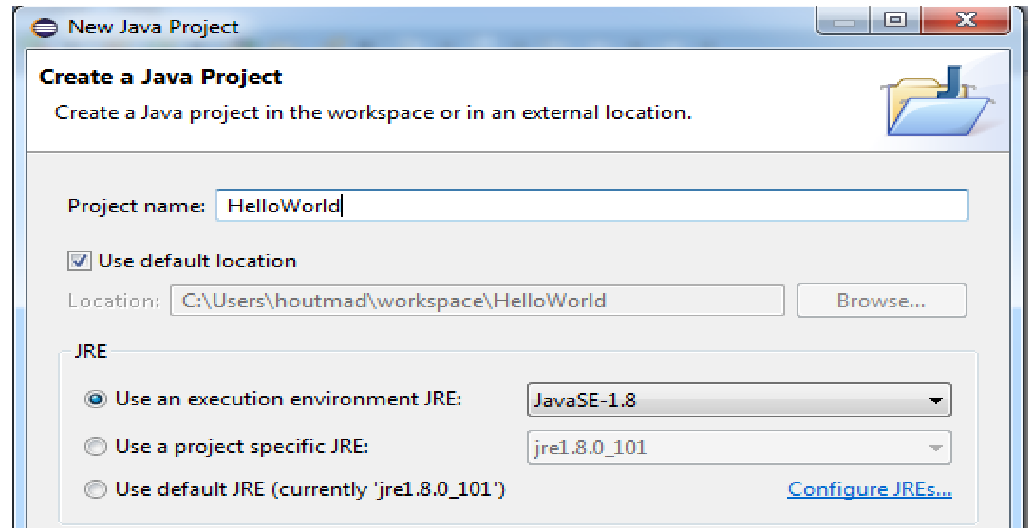
First, to create the new program, you have to perform a series of steps in Eclipse which include:

1. Creating a new Java project. This involves selecting `File >> New >> Java Project` from the menu.



2. Giving a name to the project. The name can be anything you wish. It does not need to be the same as the name of any of your classes.

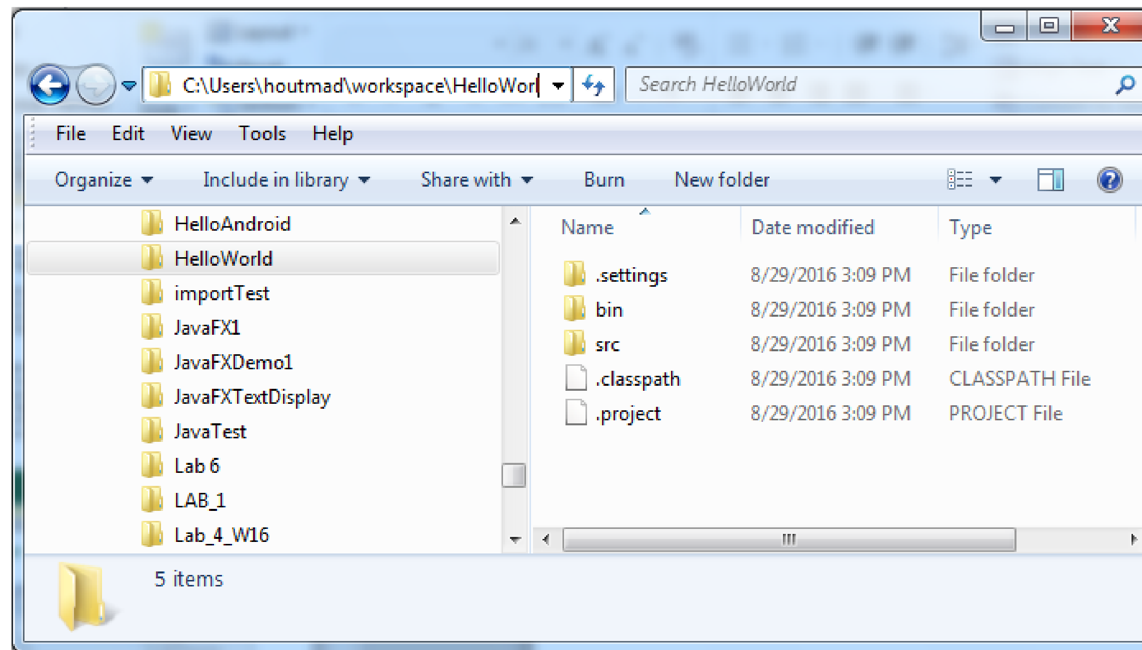
   We'll initially call our new project 'HelloWorld'.



Liang 1.12

2.1 Packages

What does this do? Creating a new project means, from Eclipse's point of view, establishing a new folder on the hard drive and populating it with certain default subfolders and files. This new folder is typically located in `C:\Users\`*`your_ user name`*`\workspace`, seen below. At this point, only the `.settings` folder holds anything, while both `bin` and `src` are empty. Additionally, the workspace folder contains two files, `.classpath` and `.project`. (Note that, in your assignments, you'll be required to zip and ship *all* of the information in the Eclipse project folder)

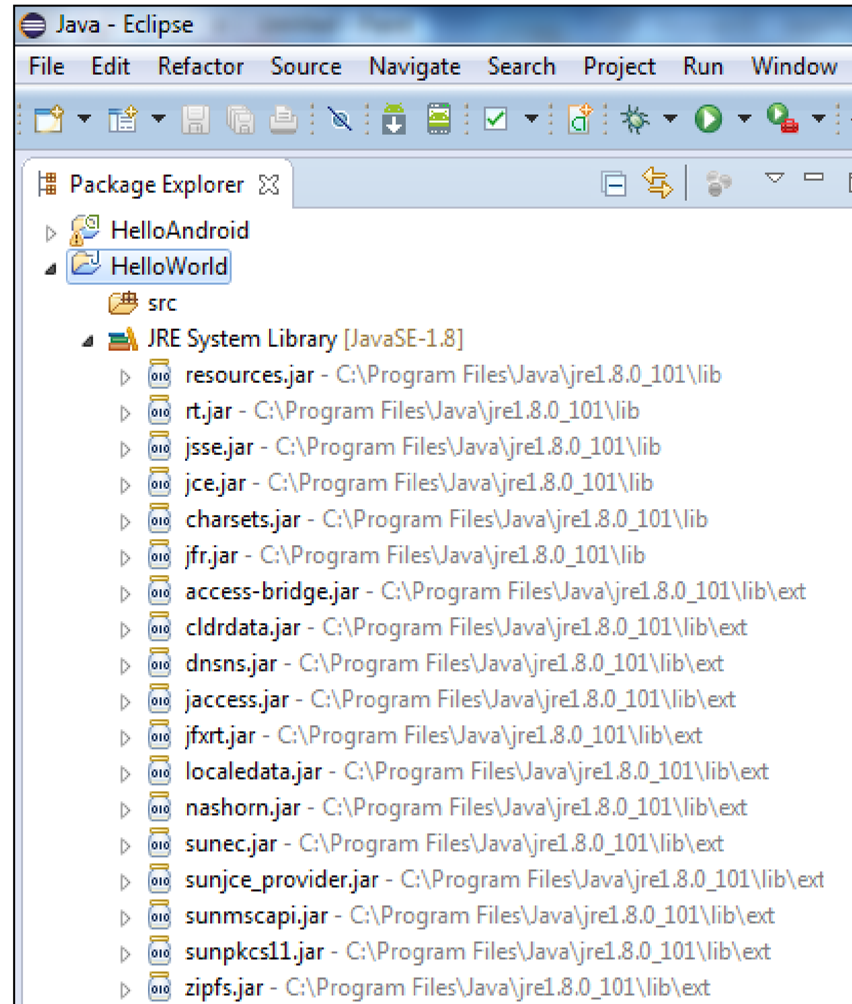


2.1 Packages

Two of the items appear in the HelloWorld folder in Eclipse:

(1) the `src` folder, shown in the previous slide in Windows File Explorer. `src` is where you'll store the .java files and **packages** that you'll be adding to your code; and

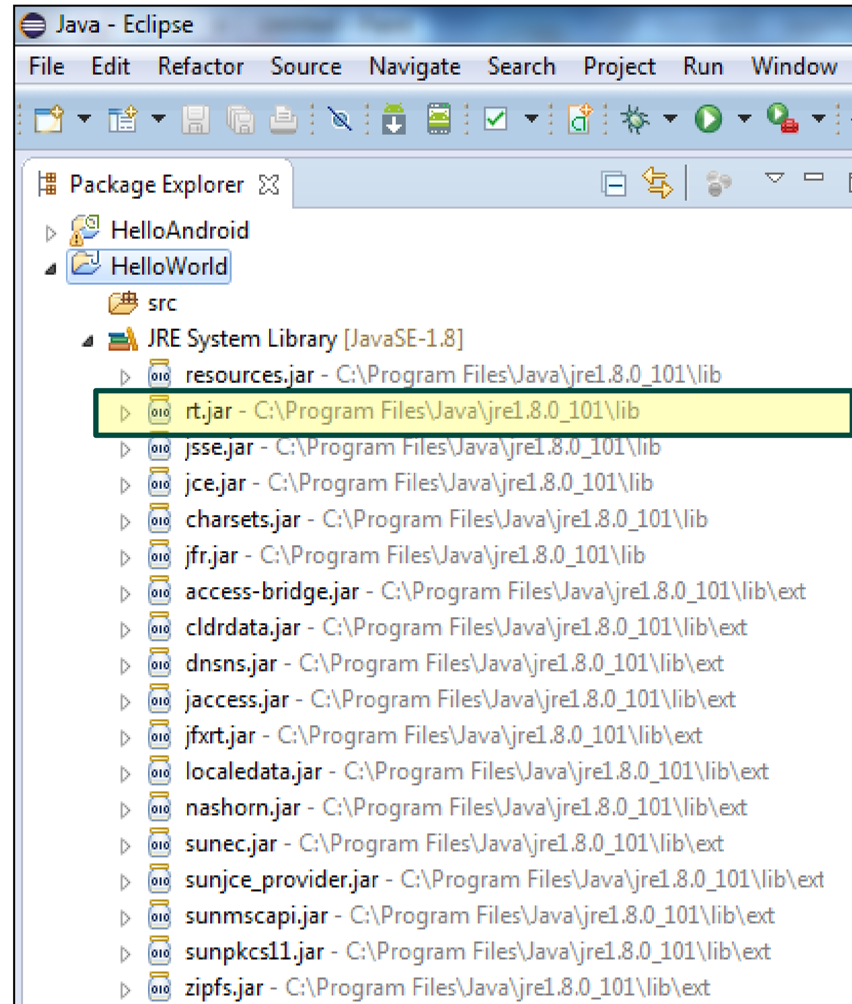
(2) the default *JRE System Library,* which contains a list of .jar files that store the default classes needed by your programs. This information is provided by the `.classpath` file, which is one of the two files loaded with the project itself, as seen in the previous slide.



2.1 Packages

Each of the .jar files in the JRE System Library is a zipped-up folder containing pre-compiled classes of code, ready to be utilized in your projects.

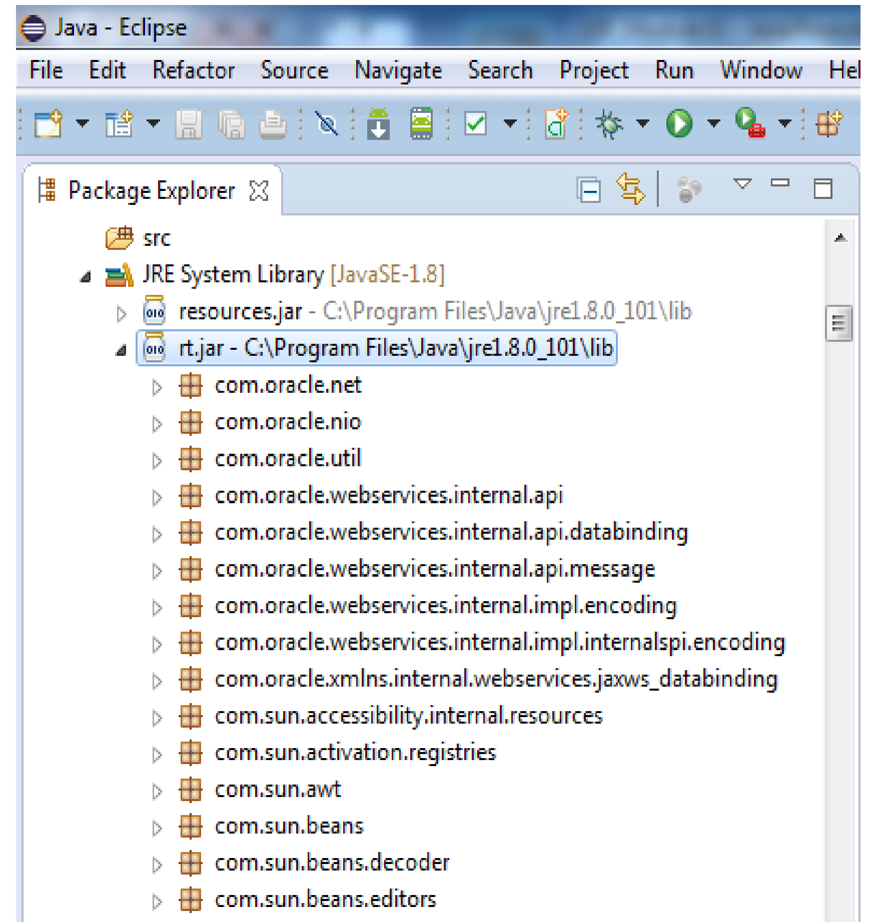
We are particularly interested in the rt.jar file, shown as the second item in the list at right. This contains, amongst other things, the default package of classes that allow your Java project to execute 'Hello World'.



2.1 Packages

Clicking on the rt.jar file—or any of the .jars for that matter—shows the list of packages the jar contains.

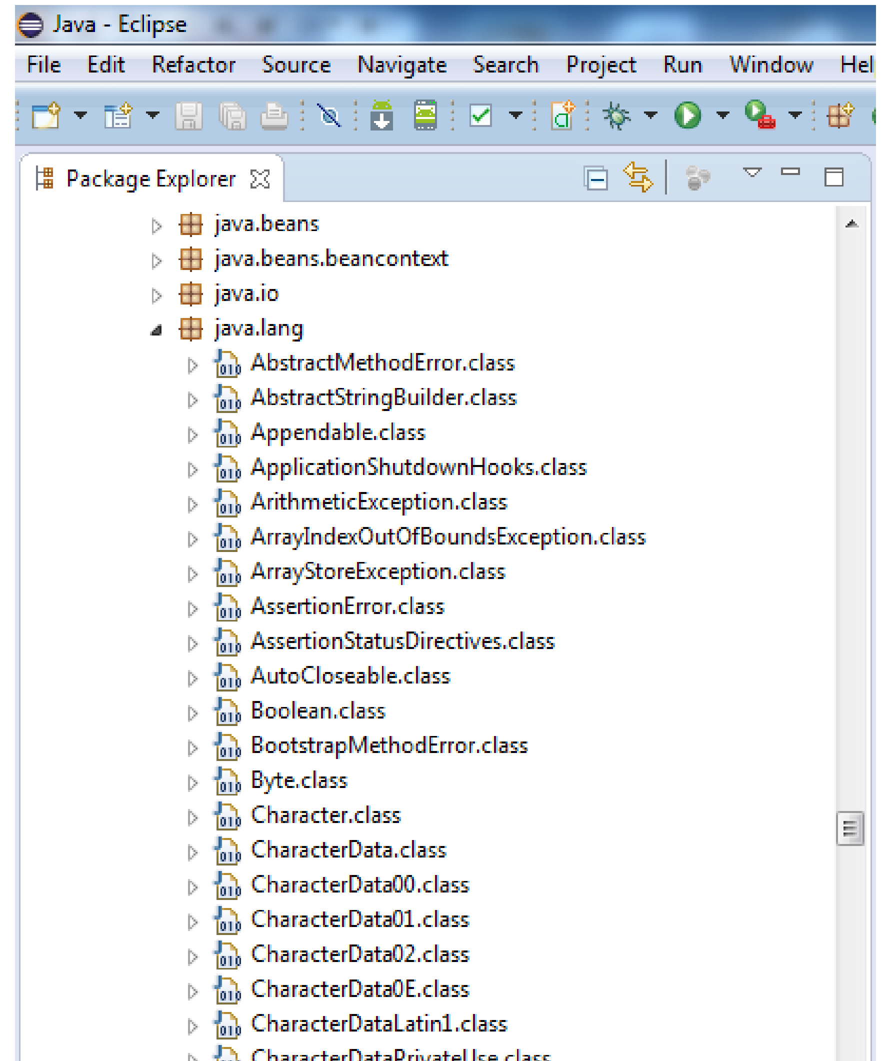
The icon for a package is a yellow box with a cross-hatch on it.



2.1 Packages

Clicking on a package icon, such as `java.lang`, indicates the various classes in the System Library that are *potentially* available to your code. Note that the contents of these packages are not installed automatically. Rather, Eclipse is showing a list of code that it is aware of, which can then be explicitly loaded by your programs, as required.

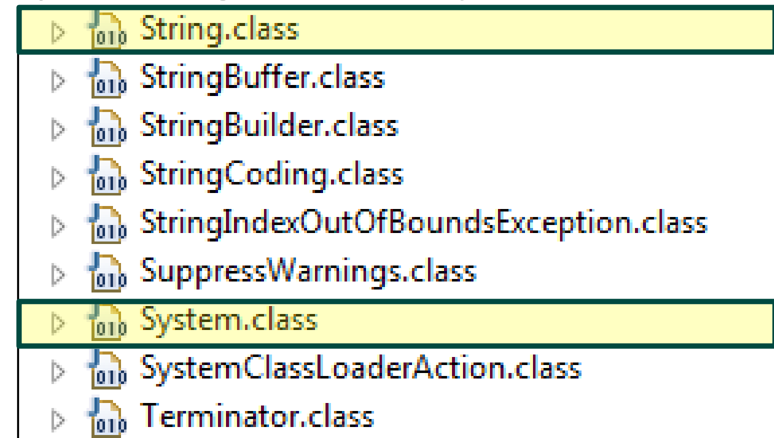
The reason for this strategy should be obvious from the number of .jar files, packages and classes: It would cost a great deal of time and memory to load *all* of the classes in *all* of the packages in *all* of the .jars. And most of these classes would never be used anyway in any given program.



2.1 Packages

While most of these classes aren't imported automatically, the classes in `java.lang` *are* immediately accessible to your program, by default. This includes the `String` class and the `System` classes, seen at right (they appear farther down the list inside the `java.lang` package of the `rt.jar` file).

java.lang (loaded by default):

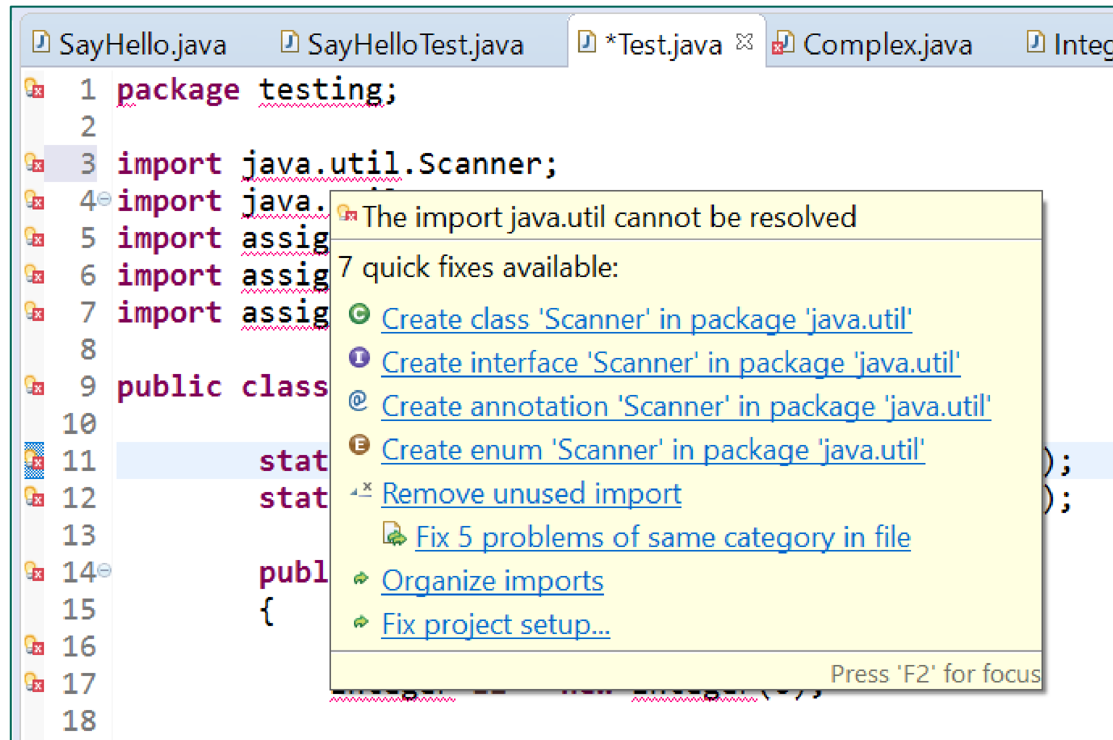


This is the reason you can use `System.out.println` 'straight out of the box' without needing to import the library: it's made available by default.

ALGONQUIN COLLEGE

2.1 Packages

Note that if the `.classpath` file is missing—perhaps you forgot to copy it—an error appears that indicates that all of your support libraries have disappeared. That's because, without a classpath, you're missing everything needed to get the program up an running, including the default `java.lang` library itself.

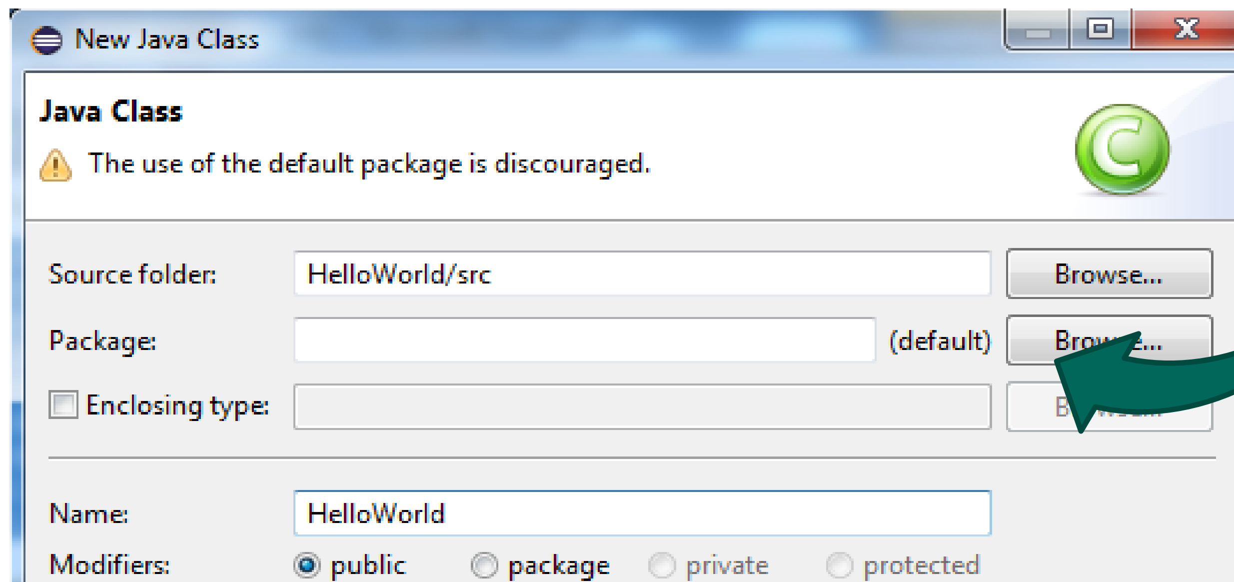


2.1 Packages

Note that the two Eclipse folders, the `src` folder and the *JRE System Library,* together correspond to something we alluded to in Module 01: that each program is a combination of built-in code, supplied with the software and found in the JRE (provided by `.classpath`), plus your own software contribution, stored in `src`.

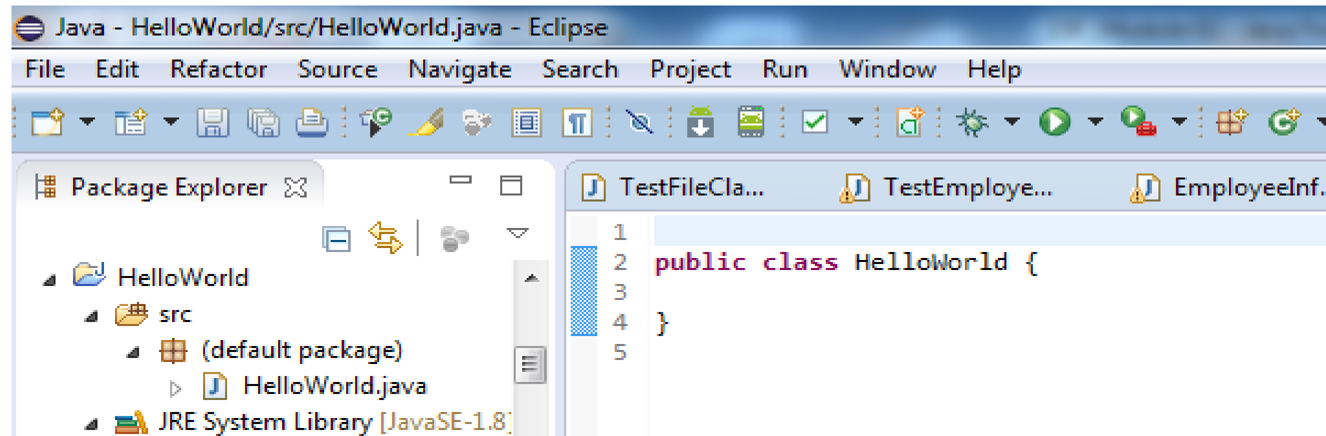
2.1 Packages

Returning to the HelloWorld project in Eclipse: once the new project is created, then, for simple programs (such as those you encountered in CST8110), you only needed to add the class that will contain the "HelloWorld" code to our default package. For this we (1) right click on the project name and select `New >> Class` from the menu (2) Enter the class name in the `Name` textbox and (3) Click the `Finish` button. If you don't enter a package name in the second textbox of the Java Class dialog window, the `(default)` package is assumed as the storage location for our .java code.

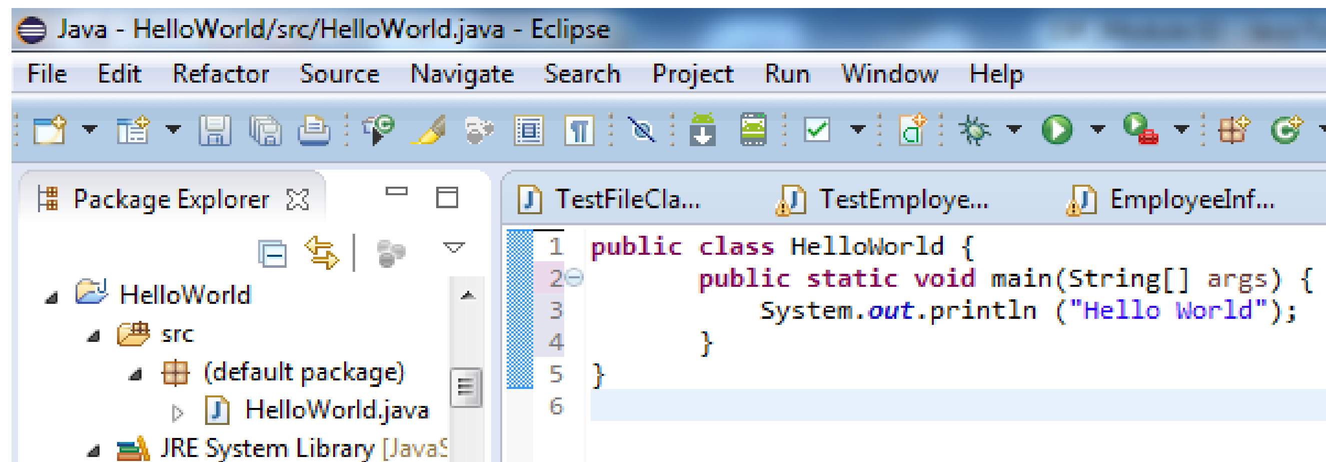


2.1 Packages

The result looks like this:

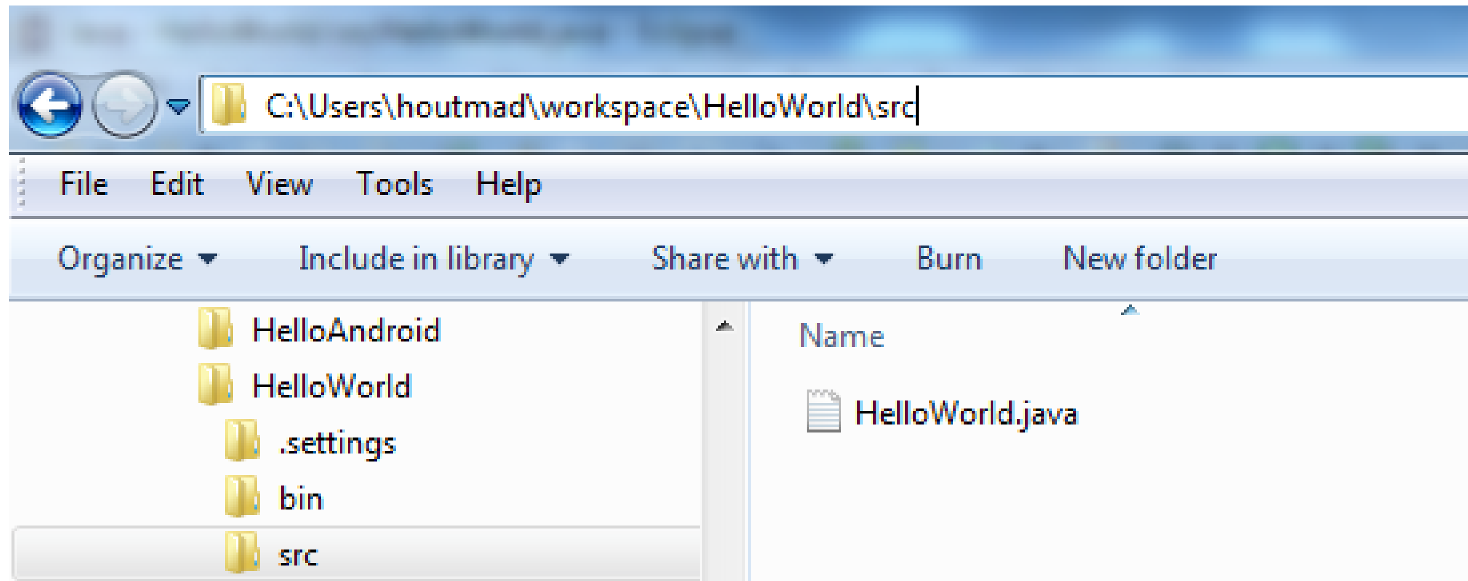


We can then insert our code into HelloWorld.java:



2.1 Packages

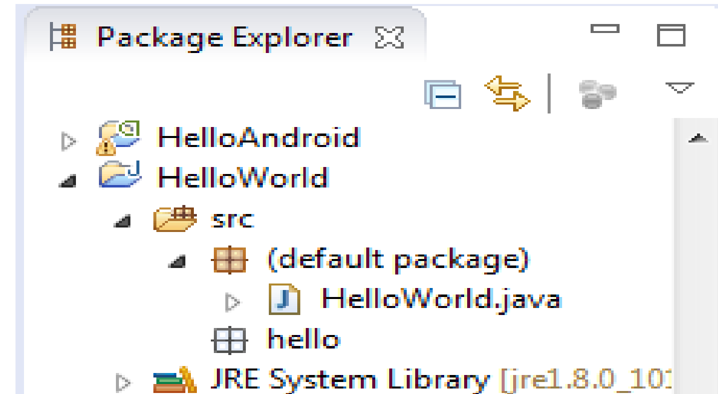
In Windows File Explorer, we can see that our new file has been added to `C:\Users\your_username\workspace\HelloWorld\src`.



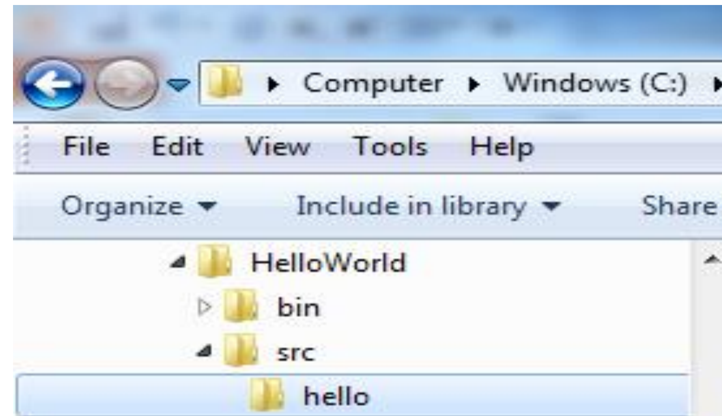
Note that there is no subfolder called 'default package' in the `src` folder; the package is what Eclipse assumes 'by default', in order to store your .java files when a specific package hasn't been specified. The words `(default package)` in the Package Manager is Eclipse's way of saying: I don't have a folder to put this in, so we'll use `src` instead

2.1 Packages

In general, use of the default package is inadvisable; your code should be stored in an appropriately-named package. In large projects, packages are essential to help organize code into logical units, as our look in the *JRE System Library* demonstrated.



To add a package to your project, right click on the project name in the Eclipse Package Manager, and select `File >> New >> Package` from the menu, and enter the new package name (called `hello` at right) in the appropriate text box.



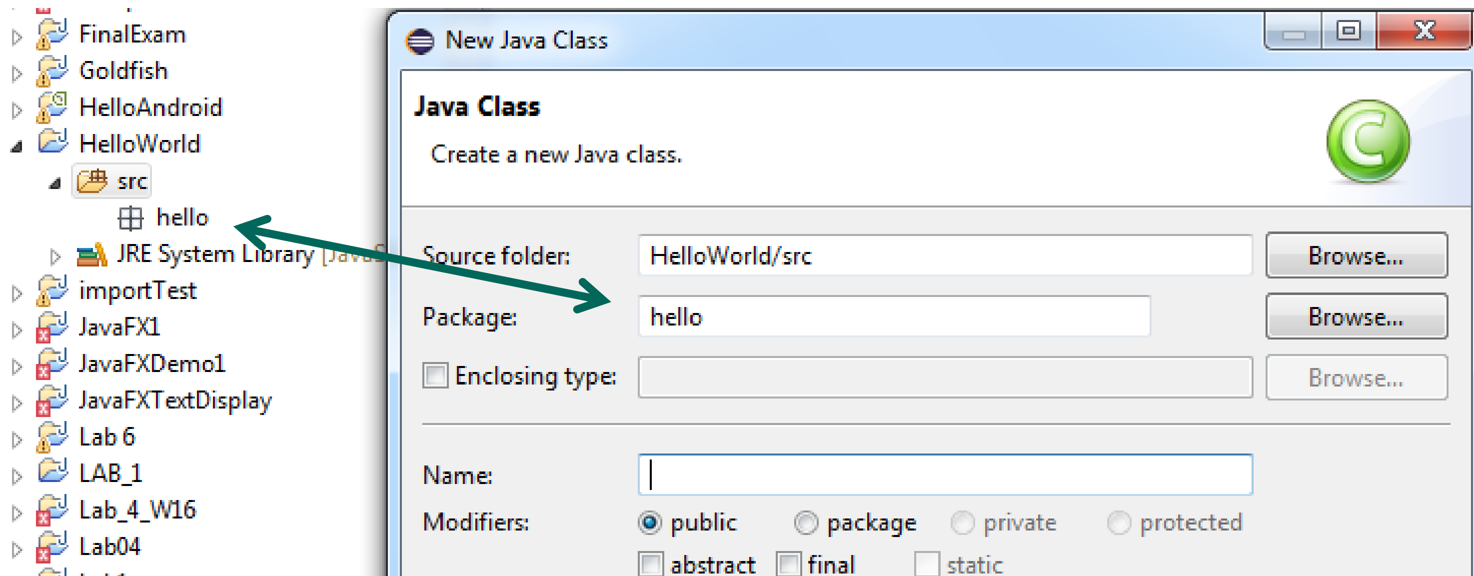
The new package shows up in both Package Manager and in Windows File Explorer (as a new folder).

Note that you can drag and drop .java files from one package into another in Eclipse.

2.1 Packages

When building a new project, it is generally advisable to add the packages *after* the project has been created, but *before* the classes; this saves you having to move .java class files around afterward.

When there's only one package in your project, Eclipse assumes that's where you want your new class to go, as seen below…

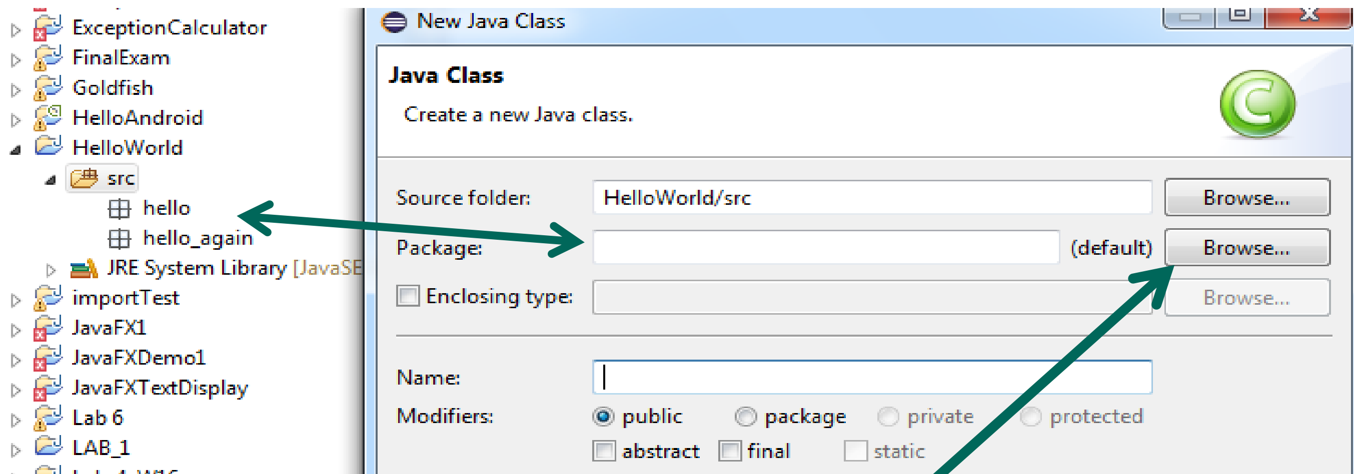


Liang 9.8

D&D 8.14

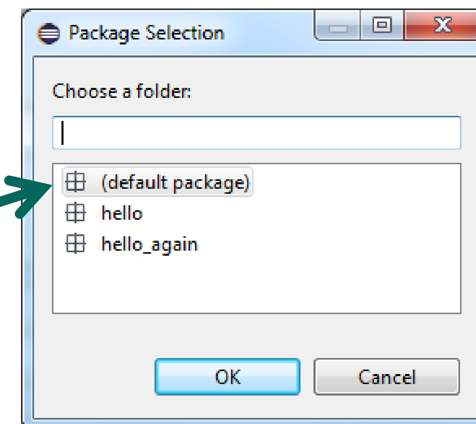
2.1 Packages

...but when more than one package is present, Eclipse doesn't hazard a guess as to which package you'd like to put the new class in.



In this case, you should select the 'Browse' button to the right of the Package textbox, and specify which of the packages the new class belongs in.

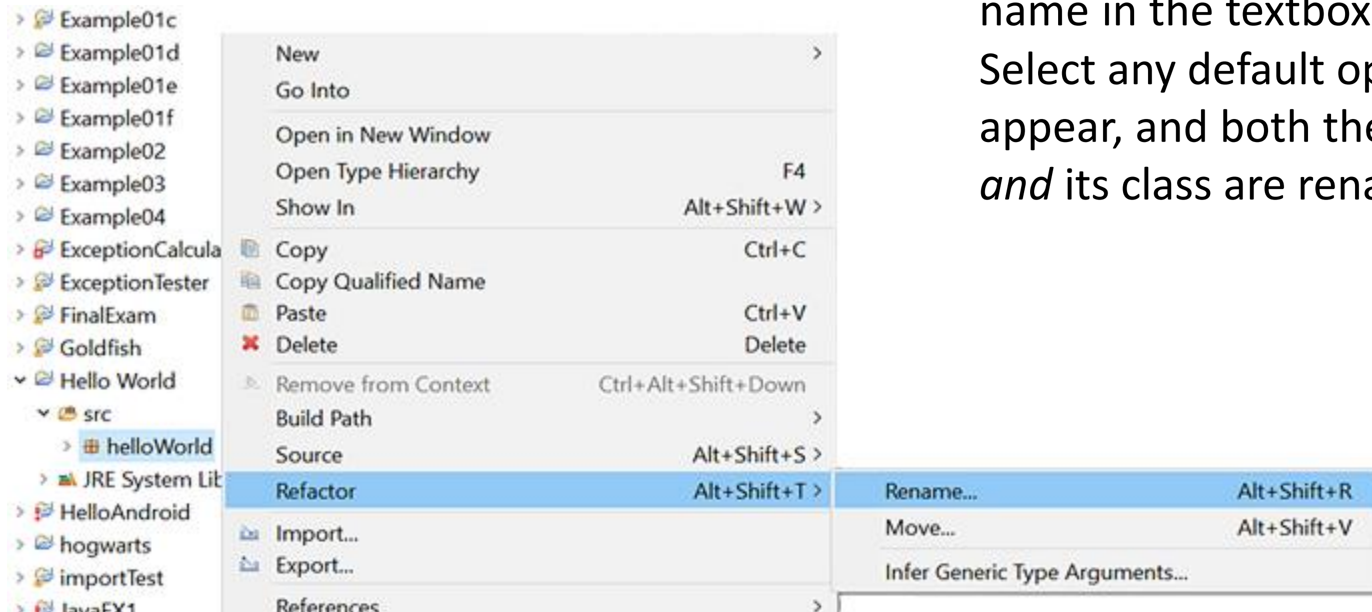
Note that the default package is still there; it 'lives' in the `src` folder (whether there's any .java files stored in `src` or not).



2.1 Packages

Finally, what if we wish to change the name of a class inside a package? We could copy the Java code, delete the original project in Eclipse, create a new project with a new class folder name, and paste the code back in. This is inefficient and fraught with possible complications. A better way is simply to **refactor** the code, that is, to change it internally while keeping its behaviour intact.

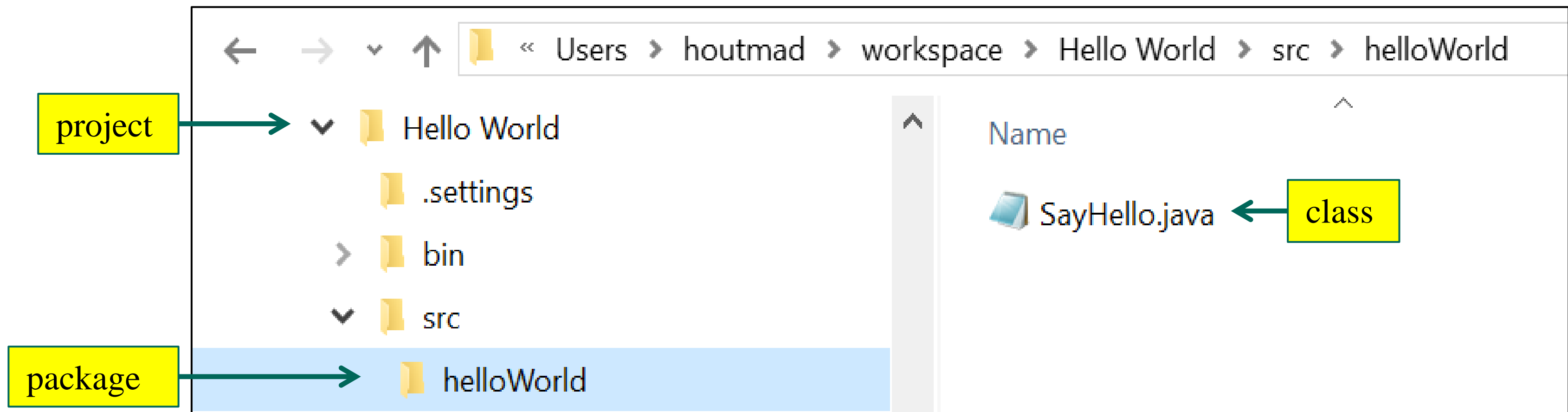
To refactor our 'HelloWorld' code in Eclipse, right click on the `HelloWorld.java` folder, and select `Refactor >> Rename…` from the menu, and supply a new name in the textbox provided. Select any default options that appear, and both the .java file *and* its class are renamed.



2.1 Packages

In our example, we'll change the file and class name to `SayHello`. The new name is reflected in the file name in Windows File Explorer (see below).

Note that if you click on the class folder first before refactoring, then both the folder and the java classname, along with any references to that class inside the code, are all updated to reflect the change. But if you start by clicking on the package name and refactor *only* it, then the underlying code is left intact. The refactoring process is smart enough to understand what needs to be refactored, and what doesn't; if there's some ambiguity, it will prompt you for clarification.



2.1 Packages – Notes

Before we continue, there are a number of things to note about packages:

1. Packages are the main form of storage for any collection of classes. Packages are essential if we are to build large programs, which potentially could contain thousands of classes and hundreds of packages.

2. The name of packages starts with a lowercase character, by convention

3. The word `package`, followed by the package name, is inserted by default at the top of every .java file contained in that package. This is essential, since all such .java files need to be aware of what other .java files share the same **namespace**. Thus every .java file in the `helloWorld` package must have

```
package helloWorld;
```

   as the very first line.

4. The package name, like the project name, can be any valid identifier. It does not have to correspond to the .java file/class name, which must be the same. But you cannot, for example, call a package `package`, since that word is reserved in Java.

2.1 Packages – Notes

5. By convention, packages are labelled according to their source URI, but listed in reverse order, starting with the top level domain name (like .com, .net. or .ca), and working backwards. For example, consider again the list of packages inside rt.jar, shown at right. The `com.sun.awt` package originates from the awt.sun.com web site.

   In keeping with this convention, we will label our packages with the prefix `cst8284`, followed by an appropriate label such as `lab1`, `assignment1`, or, in this example, `sayHello`. Hence the complete package name for this example—refactored again from our earlier version—will be: `cst8284.sayHelloExample`.

rt.jar - C:\Program Files\Java\jre1.8.0_101\lib
- com.oracle.net
- com.oracle.nio
- com.oracle.util
- com.oracle.webservices.internal.api
- com.oracle.webservices.internal.api.databinding
- com.oracle.webservices.internal.api.message
- com.oracle.webservices.internal.impl.encoding
- com.oracle.webservices.internal.impl.internalspi.encoding
- com.oracle.xmlns.internal.webservices.jaxws_databinding
- com.sun.accessibility.internal.resources
- com.sun.activation.registries
- com.sun.awt
- com.sun.beans
- com.sun.beans.decoder
- com.sun.beans.editors
- com.sun.beans.finder
- com.sun.beans.infos
- com.sun.beans.util
- com.sun.corba.se.impl.activation
- com.sun.corba.se.impl.copyobject
- com.sun.corba.se.impl.corba
- com.sun.corba.se.impl.dynamicany
- com.sun.corba.se.impl.encoding
- com.sun.corba.se.impl.interceptors
- com.sun.corba.se.impl.io
- com.sun.corba.se.impl.ior

2.1 Packages – Notes

6. Two packages are automatically loaded by Eclipse whenever a new project is created:

    a. `java.lang`
    b. `default package` (a.k.a. "the package with no name")

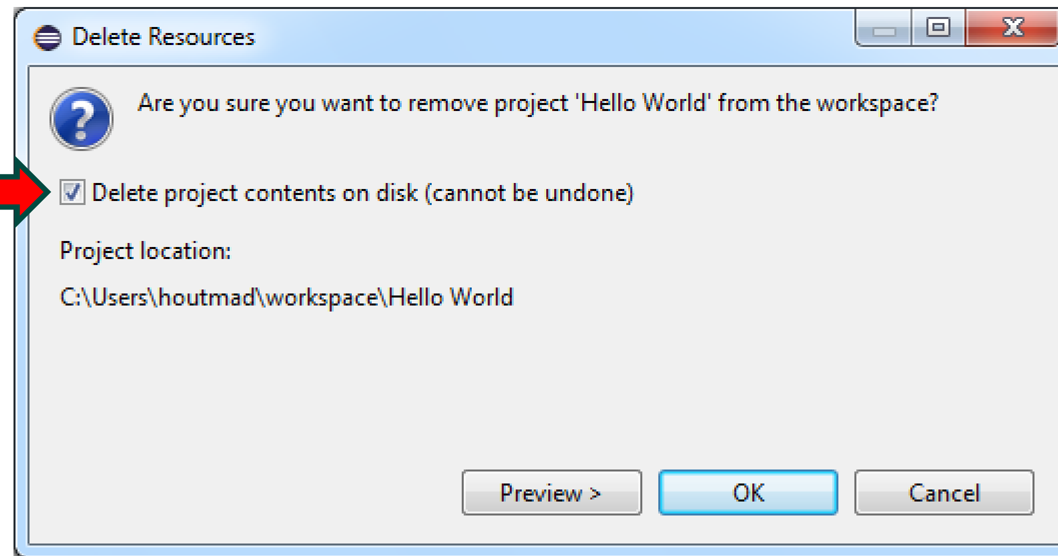
    Neither of these show up as a folder in `src`, since they are both used internally by your program. A folder is created only when *you* add a package using `File >> New >> Package` (which is appropriate, since a package, like a folder, serves to keep related material grouped together.)

7. There's a mistaken tendency to believe that when you import a class from something like `java.lang.System`, that this must represent a hierarchical folder system, with `java` as one folder, `lang` as a subfolder, and `System` as a file inside that contains the appropriate class. But as we've just seen, it's not that simple. .jar files are compressed files that can contain packages (i.e. folders) which can contain other packages and even other .jar files. But the notion that there's a hierarchical system at work is only approximately correct.

2.1 Packages – Notes

8. When deleting projects in the Package Explorer (by right-clicking on the project name and selecting `Delete` from the menu) it's important to select the 'Delete' check box, indicated below, if you wish to completely delete the project folder, and all its subfolders, *in its entirety*, from your computer. Failure to do so—the unchecked state is the default—means that residual components of the project may be left on your hard drive. These may cause problems if you attempt to create a new project with the same name as the deleted project: Eclipse will complain that it cannot create a new project of that name when such a project already exists—even though that project does not appear in Package Explorer.

Failure to completely delete a project from the workspace may lead to future complications



2.1 Packages – Notes

9. When handing in assignments, you must include all of the information related to your project. This means that both the `src`, the `.classpath`, and *all* of the files, folders and packages involved in a program: these must all be included. This can, and should, be done in the form of a single .zip or .jar file according to the instructions that will be provided to you.

2.2 Classes

A class is the conceptual starting point for any object-oriented program.

All java code is contained in a class. Class declarations start with the keyword `class` preceded by an access modifier (which, for most of the classes in this course, is limited to `public`). Each class declaration will typically be in the form:

```
public class ClassName {

   …code goes inside here…

}
```

But recall that your classes are initially stored in a .java file in `src`. Only following compilation to bytecode are they stored as .class files.

Liang 9

D&D 3

ALGONQUIN COLLEGE

2.2 Classes

Each class contains properties (or fields) and methods,* along with access modifiers that determine their visibility

```
public class ClassName {
    [public/protected/private] propertyName;

    [public/protected/private] returnType methodName(){
            …code goes here
    }
}
```

Liang 9.8

D&D 8.3

* In Java, you're not limited to just properties and methods; classes may also contain inner classes and interfaces, which will be introduced later in the course.

2.2 Classes

Private access means that a member can only be accessed by other members within the class itself.

Public access means that a member is accessible to other methods outside the class; when the class is instantiated, its public members will be available for use by other methods.

Protected access means that a subclass has access to a member in the superclass, along with any other classes in the same package. Essentially, *the member is treated as if it was public to everything inside the class and everything that inherits from the class, but private to everyone else*. We'll deal with this access modifier in more detail later in the course, when we address inheritance.

Packages essentially provide a fourth level of access modification, since the contents of one package are automatically isolated from the contents of another. When a member's access modifier is unspecified, the latter is used by default. Such members are said to be **package private**.

2.2 Classes

The `main()` method is special; it is the usual starting point for program execution. The format of the `main()` method is:

```
public static void main(String[] args){ }
```

Note that `main()` takes an array of `String` data types as arguments; these can be used at the command line to specify additional parameters for your program, as we'll see in an example below. The following two forms of this argument:

`String args[]` and `String[] args`

are equally acceptable.

D&D 2.2

2.2 Classes

Each project may contain more than one package, and each package may contain more than one .java file. Each .java file can only contain one `public` class, and the name of that class must be the same as the name of the .java file itself. Consider a package containing two class files called ClassName1.java and ClassName2.java:

package (any name)

ClassName1.java

ClassName1

...code goes here...

ClassName2.java

ClassName2

...code goes here...

2.2 Classes

D&D 3.2.3

When you compile a package having multiple classes, `javac` creates separate `.class` files based on each class name in the original file. To compile all the `.java` files in a project into `.class` files using `javac`, type:

package (any name)

ClassName1.java

ClassName1

...code goes here...

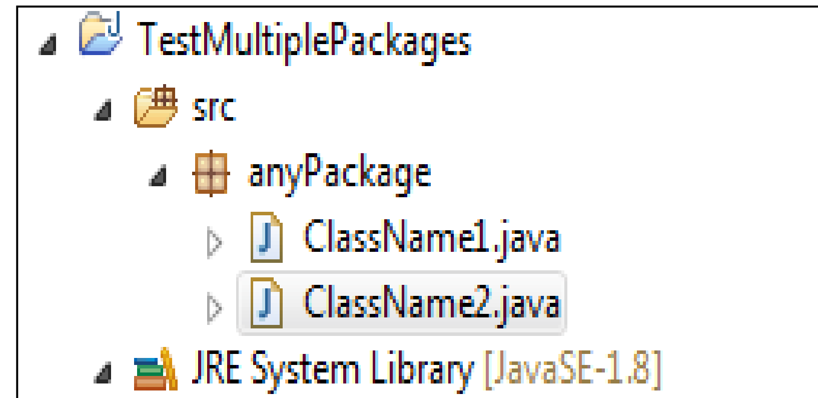
ClassName2.java

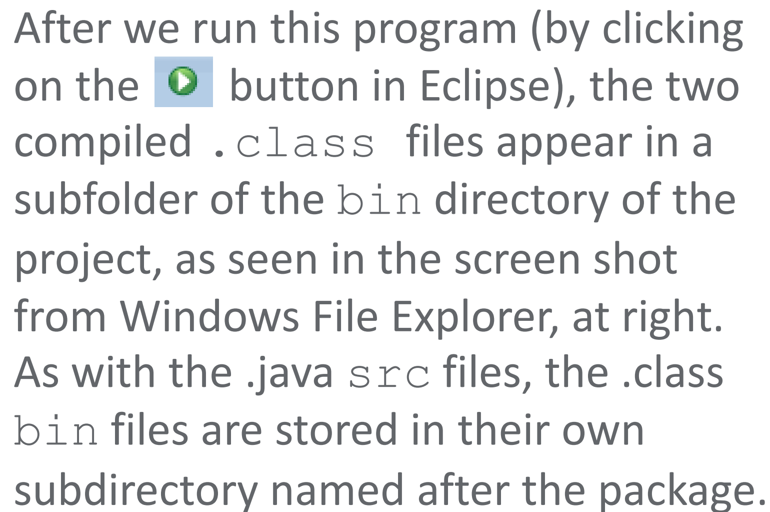
ClassName2

...code goes here...

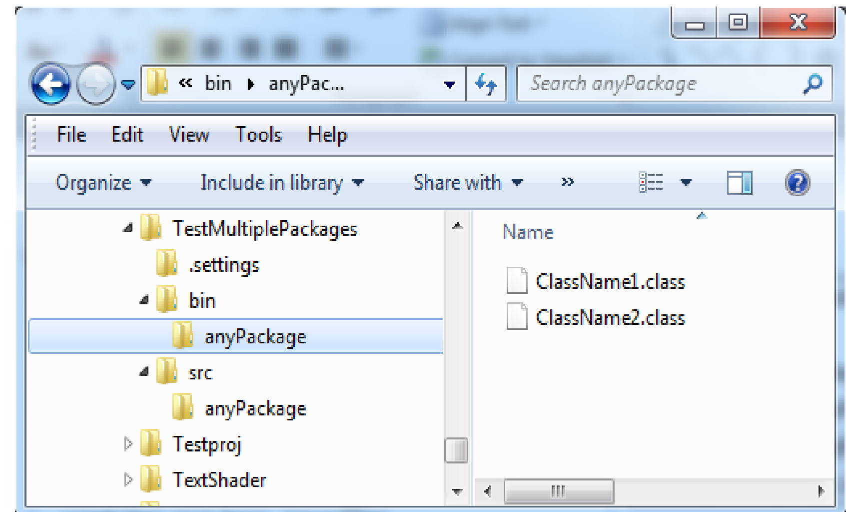
2.2 Classes

In Eclipse, running a project with multiple .java files has exactly the same effect as applying `javac *.java` at the command line. For example, say we have the following project, which contains a single package (`anyPackage`), which contains our two .java files.





After we run this program (by clicking on the button in Eclipse), the two compiled `.class` files appear in a subfolder of the `bin` directory of the project, as seen in the screen shot from Windows File Explorer, at right. As with the .java `src` files, the .class `bin` files are stored in their own subdirectory named after the package.



2.2 Classes

Although you can only have one `main()` method in each class, then, since you can have multiple .java files, you can have multiple `main`s in each package, which must be in the one public class found in each file.

package (any name)

ClassName1.java

ClassName1

```
public static
   void main()…
```

ClassName2.java

ClassName2

```
public static
   void main()…
```

2.2 Classes

You can call a *specific* `main()` method by citing its `ClassName`, i.e. by calling it using `java ClassName` at the command line. So, using the example above, typing `java ClassName1` loads and executes `main()` starting from `ClassName1`. Similarly, in Eclipse, setting the insertion point in one `.java` file or the other indicates which `main()` you wish to run initially.

package (any name)

ClassName1.java

ClassName1

```
public static
   void main()…
```

ClassName2.java

ClassName2

```
public static
   void main()…
```

2.2 Classes

It is *not* absolutely essential that there be a `main()` method *somewhere* in a Java project: there are other ways to start a program. For example, as we'll see, JavaFX graphical applications are started using the `launch()` method, rather than `main()`. However, it is good practice to always include a `main()` method in a class, since some IDEs may not work properly otherwise.

D&D 25.5.2

package (any name)

ClassName1.java

ClassName1

...no main() methods here...

ClassName2.java

ClassName2

...no main() methods here...

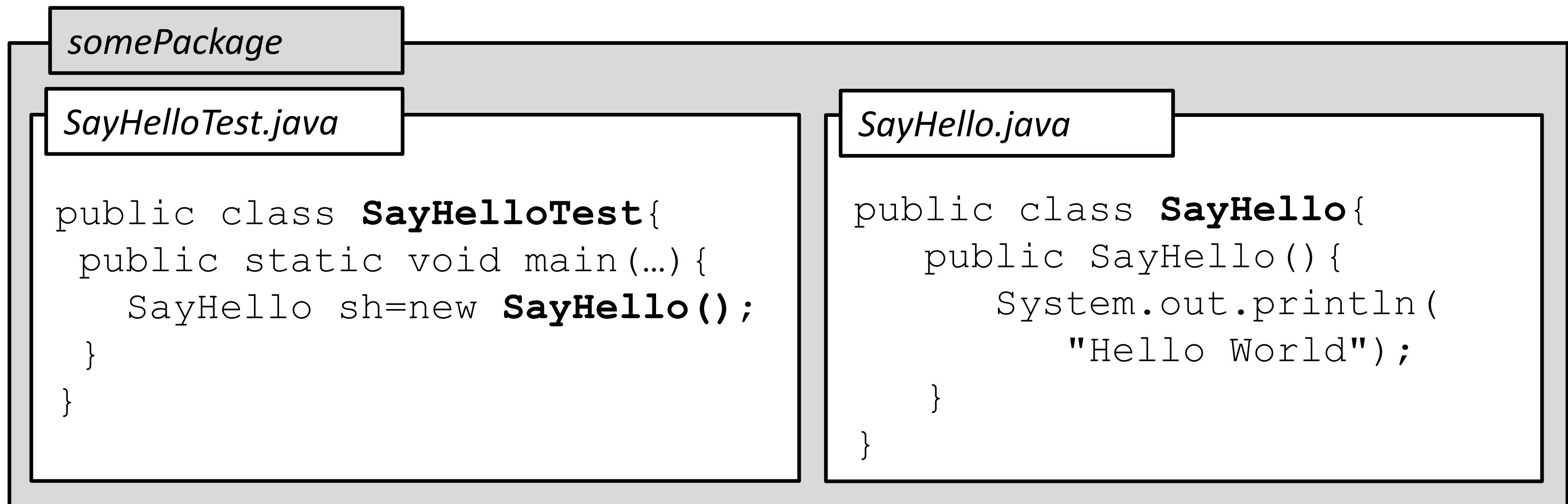
2.2 Classes

Additionally, we may have multiple packages in a project. They don't *all* need to have a `main()` method in them, since some, perhaps most, of the packages may be acting in a supporting role to the other classes found in other packages. But for non-JavaFX programs you will certainly need to have at least one `main()` method *somewhere* in the project.

ALGONQUIN COLLEGE

2.2 Classes

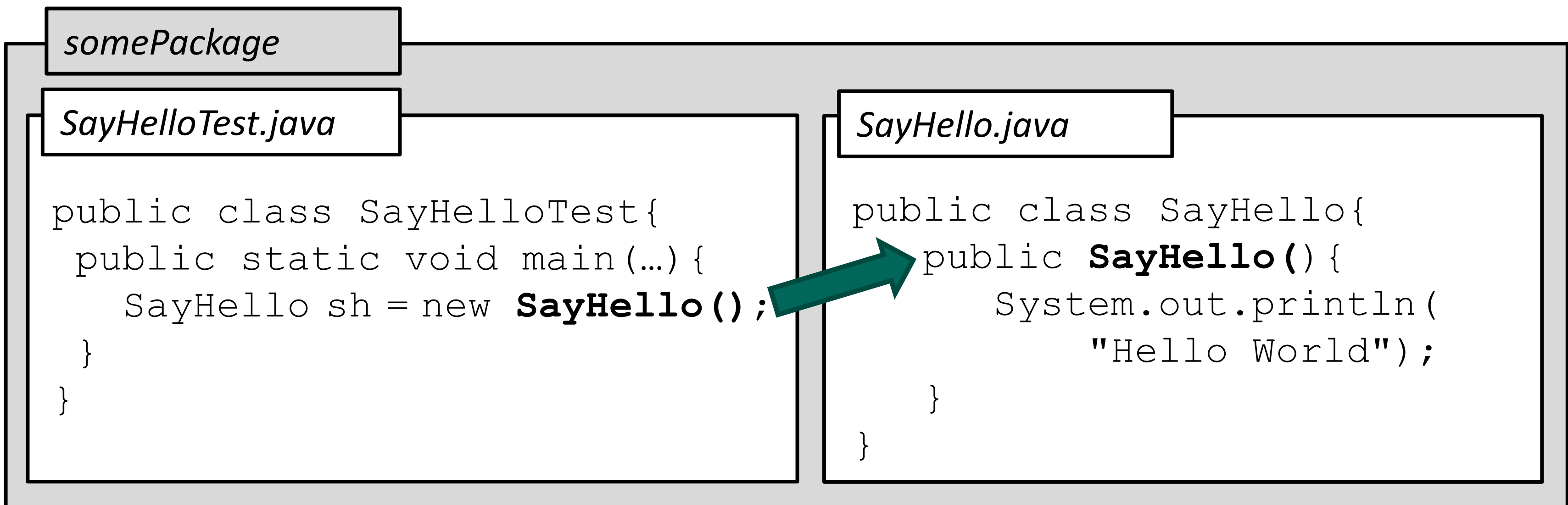
In most applications, there will be only one `main()` starting point for program execution. Typically, the class containing `main()` acts as the main 'control centre' for the code. The real work is typically off-loaded to the various helper/service classes.



Applying this methodology to our simple "Hello World" code, we'll split our original code into two classes in the same project, as indicated above. While this is admittedly overkill for such a simple project, it's importance will become apparent when we deal with larger projects later on in the course.

2.3 Constructors

In this version of "HelloWorld", the `main()` method is located in `sayHelloTest.java`. It contains a single line of code, which instantiates a new `SayHello` object



Liang 9.4

D&D 8.5

ALGONQUIN COLLEGE

2.3 Constructors

`SayHello()` is a special kind of method called a **constructor**. Typically, a constructor provides a shortcut way of instantiating a new object with certain features set by passing values in the parameter list. Here, we use it only to output a message to the screen.

The constructor *must* have the same name as the class, which of course has the same name as the .java file in which it is found.

somePackage

***SayHello**.java*

```
public class SayHello{
   public SayHello(){
     System.out.println("Hello World");
   }
}
```

2.3 Constructors

Thus the line

```
SayHello sh = new SayHello();
```

loads the new `SayHello()` object in memory, outputing "Hello World“ to the console.

Note that, since we're not using the variable `sh` for anything, we really don't need to declare it. The code in `main()` above will work identically if we write:

```
public class SayHelloTest{
    public static void main(…){
        new SayHello();
    }
}
```

2.3 Constructors

When any two class members (including constructors, which are a special kind of method) have the same name, they are considered to be **overloaded**. For example, our `SayHello` class might contain three different `SayHello()` methods, as follows:

somePackage

***SayHello**.java*

```
public class SayHello{

    public SayHello(){
        System.out.println("Hello World");
    }
    public SayHello(String s){
        System.out.println("Hello " + s);
    }
    public SayHello(String[] strAr){
        // …etc.
    }
}
```

Liang 6.8

D&D 6.12

2.3 Constructors

Which constructor gets executed depends upon the **signature** of constructors available.

The signature consists of:

1. The name of the method
2. The number of parameters
3. The type of each parameter
4. The order of the parameters

Note that the signature does *not* include a method/constructor's *return type*, nor does it include the *name* of any identifier in the parameter list—a fact that becomes more important when we look at inheritance in the next module.

Which one of the three `SayHello()` constructors will be used depends upon the signature that best matches the one used in the calling program.

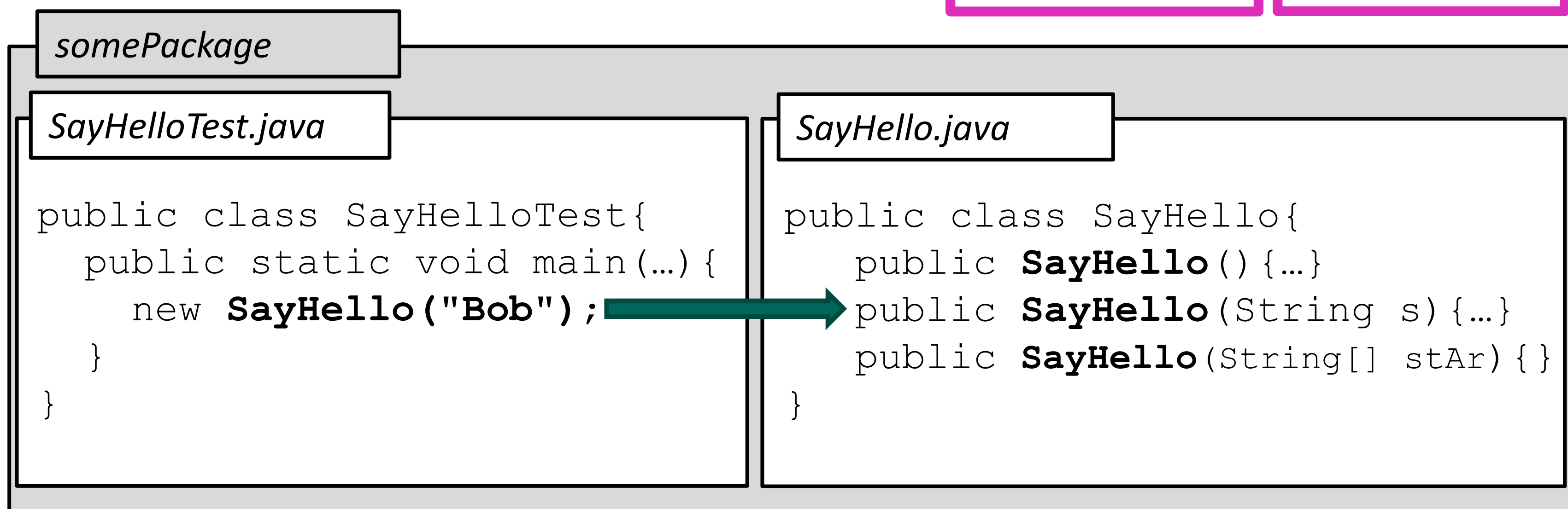
2.3 Constructors

For example, if, from within `main()` we call

```
new SayHello();
```

then the first constructor in the `SayHello` class, the **no-arg constructor**, is called. If a single String is passed to `SayHello()`, then the signature matches the second constructor, and so this constructor is invoked in the `SayHello` class, as shown below. Finally, if an array of type `String` is passed as the argument of `SayHello()`, the third constructor is invoked.

Liang 11.3.2 | D&D 8.6



2.3 Constructors

Overloaded constructors, as well as overloaded methods, are frequently **chained** together. This means that one, less-powerful constructor, uses the code of another, more-powerful constructor, to do its job for it.

Let's see how this would apply to our `sayHello()` class. Notice that the first two `SayHello()` constructors do essentially the same job, printing out a string. But the second constructor takes a `String` as the argument, while the first is empty.

somePackage

***SayHello**.java*

```
public class SayHello{
    public SayHello(){
        System.out.println("Hello World");
    }
    public SayHello(String s){
        System.out.println("Hello " + s);
    }
}
```

2.3 Constructors

Rather than reproduce *almost* the same code twice (once in each constructor), we can call the second constructor from within the first constructor. The no-arg constructor offloads the actual business of outputting the string to the second constructor, which does most of the 'heavy lifting'.

somePackage

SayHello.java

```
public class SayHello{

    public SayHello(){
        new SayHello("World");
    }

    public SayHello(String s){
        System.out.println("Hello " + s);
    }
}
```

Inadvisable: use `this` rather than instantiate a new object

2.3 Constructors

Rather than invoke two new objects in `SayHello`, a more efficient way is to use the word **this**. `this` *represents the current object*. The code is shown below.

somePackage

***SayHello**.java*

```
public class SayHello{

    public SayHello(){
        this("World");
    }

    public SayHello(String s){
        System.out.println("Hello " + s);
    }
}
```

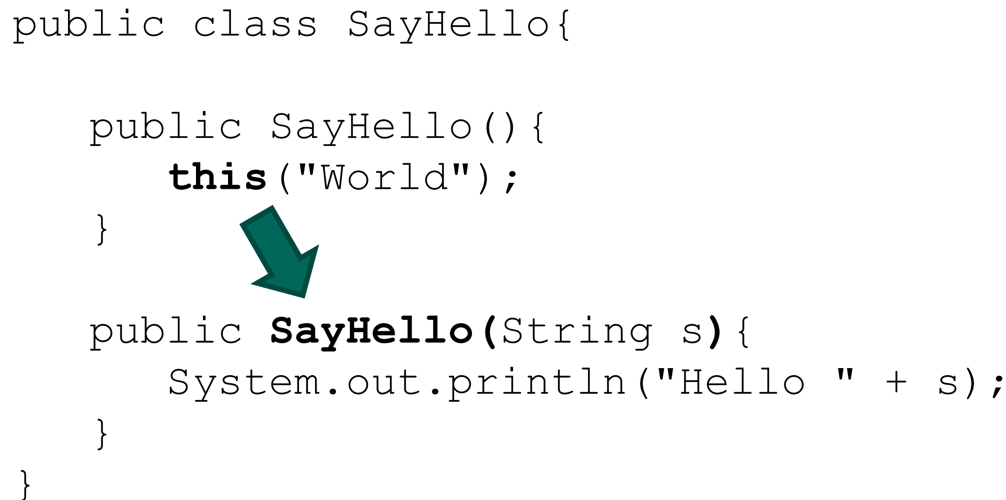
Liang 9.14 **D&D 8.6**

2.3 Constructors

A useful trick: whenever you see `this`, mentally substitute the name of the class. Thus

`this("World")` becomes `SayHello("World")`

which clearly is intended to invoke the `SayHello(String s)` constructor.




```
public class SayHello{

    public SayHello(){
        this("World");
    }

    public SayHello(String s){
        System.out.println("Hello " + s);
    }
}
```

2.3 Constructors

Recall that we included a third constructor in our code a few slides back. This was designed to read in an array of strings. Let's look at how we might output the contents of this array to output multiple "Hello" statements, and then chain this constructor in with the other two constructors. The `SayHello()` code needed to display the contents of the array is:

somePackage

SayHello.java

```
public class SayHello{
   public SayHello(String[] strAr){
      for (int i=0; i < strAr.length; i++)
         System.out.println("Hello " + strAr[i]);
   }
   //… the other constructors go here
}
```

2.3 Constructors

We can chain the second constructor to the third using a trick that converts individual strings to an array of strings (in this case, an array with only a single item).

****Note! An array of `String`s is an `Array` object; it is not a `String` object. Hence the last two constructors take distinctly different data types as parameters.

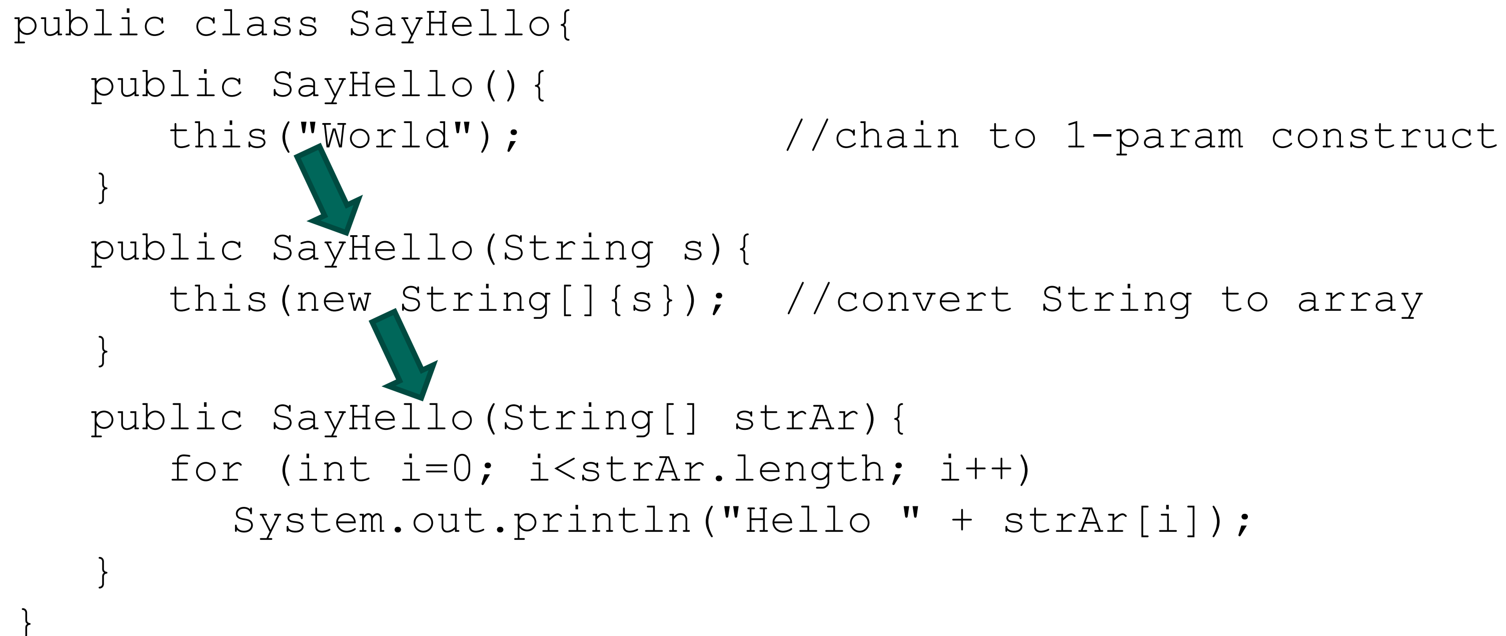
somePackage

SayHello.java

```
public class SayHello{
   public SayHello(){
      this("World");            //chain to 1-param construct
   }
   public SayHello(String s){
      this(new String[]{s});  //convert String to array
   }
   public SayHello(String[] strAr){
      for (int i=0; i<strAr.length; i++)
         System.out.println("Hello " + strAr[i]);
   }
}
```

2.3 Constructors

Note how the three overloaded constructors are chained together. The no-arg constructor calls the 1-parameter constructor; the 1-parameter constructor calls the array constructor. The third constructor does all of the real work.




```
public class SayHello{
    public SayHello(){
        this("World");             //chain to 1-param construct
    }
    public SayHello(String s){
        this(new String[]{s});  //convert String to array
    }
    public SayHello(String[] strAr){
        for (int i=0; i<strAr.length; i++)
            System.out.println("Hello " + strAr[i]);
    }
}
```

2.3 Constructors

We are of course not limited to running only a single constructor in our test script, as the modified `sayHelloTest` code shows below:

somePackage

SayHelloTest.java

```
public class SayHelloTest {
   public static void main(String[] args){
      new SayHello("Bob");
      new SayHello("Mary");
      new SayHello();
      new SayHello(args);  // args input at command line
   }
}
```

2.3 Constructors

The complete, final version of this iteration of 'Hello World' is shown below.

In `SayHelloTest.java` we will have:

```
package cst8284.sayHelloExample

public class SayHelloTest {
    public static void main(String[] args){
        new SayHello("Bob");
        new SayHello("Mary");
        new SayHello();
        new SayHello(args);
        // …other constructors and the 'main' business
    }
    // …private methods and fields go here
}
```

2.3 Constructors

In the SayHello.java file we will have:

```
package cst8284.sayHelloExample
public class SayHello {
    public SayHello(){
        this("World");      //chain to 1-param construct
    }
    public SayHello(String s){
        this(new String[]{s});   //convert String to array
    }                        // and chain to next constructor
    public SayHello(String[] sAr){
        for (int i=0; i<sAr.length; i++)
            System.out.println("Hello " + sAr[i]);
    }
}
```

ALGONQUIN COLLEGE

2.3 Constructors – Notes

You may reasonably be wondering if chaining constructors and methods in this fashion is worth the extra time spent in development, testing, and execution. Certainly this approach is more costly than writing single, stand-alone versions for each constructor? Yes, it is. But…

2.3 Constructors – Notes

…the advantages of this technique far outweigh its disadvantages. These include:

a) *Ease of Maintenance*: Changing the output—say "Hello" to "Hi" in the `SayHello` class—doesn't require that you change every constructor, just the final one. (Imagine having a dozen unchained constructors…);

b) *Code Reuse*: Once you have one of your methods working, why reinvent the wheel again and again? Simple chain to it;

c) *Less Chance to Introduce New Errors during Upgrades*: when changes are needed, there is less code to alter, hence less chance of introducing mistakes. You only need to upgrade one constructor in the chain, not all of them;

d) *Consistent Usage*: by chaining your constructors, all constructors behave in a predictable fashion, because they are all, ultimately, based upon the same base constructor. The client of such a class understands this implicitly, and doesn't have to relearn each constructor separately, since they can be relied upon to all behave in the same, intuitive manner. (We'll see this later when we look at the various ways `Scanner()` is used.)

2.3 Constructors

In the example below, the `SaveEmployeeInfo` class is the assumed to sit at the front end of a program designed to save an employee's first name, last name, and middle initial into an online database, based on some reasonable assumptions about how the information was initially entered.

As a first step, the employee's full name and/or employee number is obtained and three overloaded constructors are available to handle this information, which may be incomplete. We'll assume that if only the employee number is available, then the full name can be pulled off a spreadsheet.

The third of the of the three constructors, the one the previous two are linked to, then uses an overloaded, chained method to handle three different variations on the way a person's name might be entered, i.e. as last-name only, first-middle-last, or first-last.

2.3 Constructors

```
public class SaveEmployeeInfo{
   public SaveEmployeeInfo(){} // no-arg constructor

   public SaveEmployeeInfo(long empNum){
      this(empNum, XL.getEmployeeNameFrom(empNum));
   }
   public SaveEmployeeInfo(String fullName){
      this(0, fullName);
   }
   public SaveEmployeeInfo(long empNum, String fullName){
      String[] sAr = fullName.split(" ");
      final int LAST = sAr.length - 1;
      switch (sAr.length){
         case 0:  System.out.println("Error: No name"); exit(0);
         case 1:  setFullName(sAr[0]); break;
         case 2:  setFullName(sAr[0], sAr[1]); break;
         case 3: case 4: case 5: case 6: case 7: case 8:
                  setFullName(sAr[0], sAr[1], sAr[LAST]); break;
      }
   } …
```

ALGONQUIN COLLEGE

2.3 Constructors

```
  // one string; assume last name
  private void setFullName(String last){
     this.setFullName("",last);
  }

  private void setFullName(String first, String last){
     this.setFullName(first, "", last);
  }

  private void setFullName(String first,
                                    String middle, String last){
     this.setFullName(first, middle.toUpperCase().charAt(0), last);
  }

  private void setFullName(String first, char midInit, String last){
     DBEngine.setEmployeeName(first, midInit, last);
  }
}
```

ALGONQUIN COLLEGE

2.3 Constructors

Note the structure of the chained, overloaded constructors and methods in this example

```
SaveEmployeeInfo(long empNum)
SaveEmployeeInfo(String fullName)
SaveEmployeeInfo(long empNum, String fullName)
setFullName(String last){
setFullName(String first, String last)
setFullName(String first, String middle, String last)
setFullName(String first, char midInit, String last)
```

2.3 Constructors – Q & A

Q. What happens if I attempt to call an overloaded method, but use the wrong data types in the parameter list when I make the call?

A. The same thing happens here as normally happens whenever you pass, say, and integer value to a double data type. ***Automatic type conversion*** is used to best match a called, overloaded method to its signature. For example, consider the signature of a method that determines the maximum of two numbers:

```
public double max(double num1, double num2){…}
```

If the user then calls `max()` in an instantiated object using a `double` and an `int` in the parameter list

```
double m = myObj.max(1.0, 2);
```

then automatic type conversion is used to promote the integer—the second parameter—up to a `double`.

2.3 Constructors – Q & A

However, **ambiguous overloading** can occur if the JVM can't choose between overloaded methods. For example, say we have two overloaded methods of the above `max()` method:

```
public double max(int num1, double num2){…}
public double max(double num1, int num2){…}
```

Now if the user calls

```
double m = myObj.max(1, 2);
```

then the JVM flags an error: it can't determine which of the two overloaded methods to call upon, since each is as well, or as poorly, suited as the other.

2.3 Constructors – Notes

1. A constructor must have exactly the same (capitalized) name as the class in which it is found. Otherwise it's not a constructor, it's a class method.
2. When `this` is used to chain to another constructor, it must appear on the first line of a constructor in which it is used; no other code can come before it.
3. A **copy constructor** takes as its argument, an object, and makes a new copy of it (a subject we'll postpone to a later date.)
4. The signature is not dependent upon the identifier used in the parameter list, just its type. This somewhat trivial fact is useful to know in the next module, when we discuss subclasses and overriding.

2.3 Constructors – Notes

5. Whenever a constructor is not explicitly defined, then *every* time you create a new object, Java creates a no-arg constructor with an empty body. This constructor, called a **default constructor**, is supplied automatically, but *only if there are no other constructors in the class*.

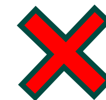
   As we'll see later, this can lead to potential problems. For now, follow this rule: whenever *any* constructor is present in a class, be sure to supply your own no-arg constructor, even if it has an empty body and does nothing.

2.3 Constructors – Notes

6. Methods have return type; constructors do *not*. Its

```
public HelloWorld()
```

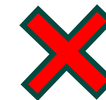
and *not* (a common mistake)



```
public void HelloWorld()
```

No return type for constructors

Therefore, you cannot treat a constructor method the same as an instantiated method; you cannot write



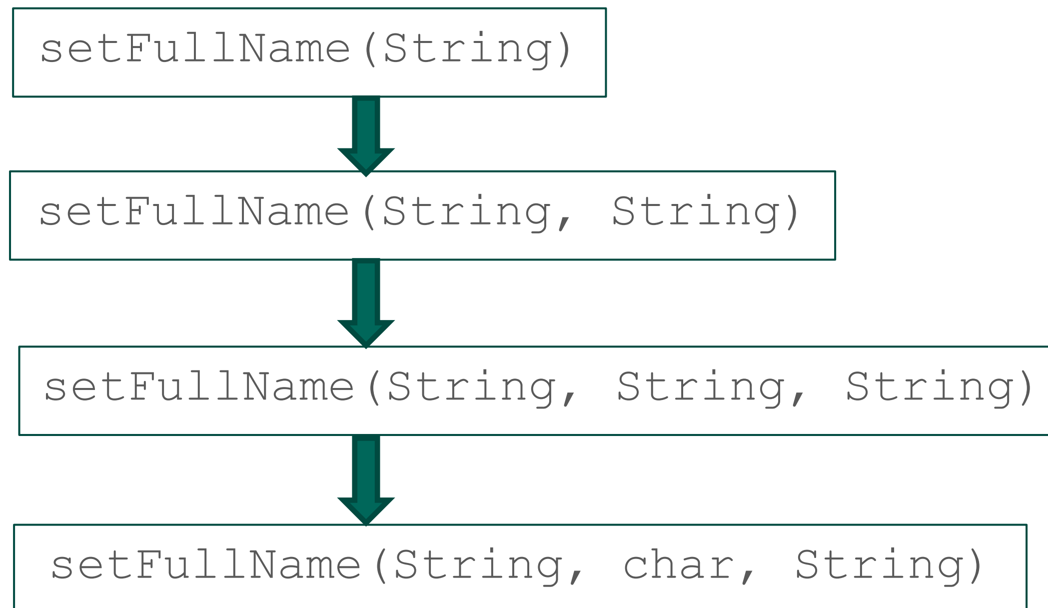
```
SayHello sh = new SayHello();
sh.SayHello(); // call default again?
```

Cannot treat constructor as a method

since a 'normal' instance method requires a return type: constructors in Java are special, and are only 'activated' when the object is instantiated with `new`; they cannot be 're-executed' afterward the way a normal method can be re-used.

2.3 Constructors – Notes

7. Chain constructors and methods from the narrow to the broad, i.e. from those with fewer parameters to those with more parameters, and from the unspecialized to the increasingly more specialized. Consider the chained `setFullName()` method(s):



2.3 Constructors – Notes

8. So far we've only looked at `public` constructors. Can a constructor be `private`? Yes! Consider the situation in which you didn't want to make the third `SayHello` constructor code `public`, but needed to chain to it internally.

   For example, say we do not wish to make the 'array of strings' constructor available for use outside the `SayHello` class. Then the `SayHello` class would be written:

   ```
   public SayHello(){…}
   public SayHello(String s){…}
   private SayHello(String sAr[]){…}
   ```

   Now only the first two constructors are available for use, but both rely on the third, private constructor to do their work for them.

Questions

1. If you store your code in the default package rather than create your own package, then, after you've run your code the first time, in which folder would you expect to find the .class files?
2. Identify the errors in the following code for the class `Cars`.

```
public class Cars{
    public void Car(){
        this("Honda")
    }
    public void Car(String make){
        this(make, "Civic");
    }
    public void Car(String make, String model){
        if ((make !="") && (model != "")) this(make, model, 0);
    }
    public void Car(String make, String model, int year){
        if (year < 1900) {
          make=make; model=model; year=year;
        } else System.out.println("Invalid year");
    }
}
```

```
public class TestCars {
  public static void
    main(String[] args){
     new Cars("Ford","Fiesta");
  }
}
```

Questions

3. Constructors do not *specify* a return type, not even `void`. But that doesn't mean they don't actually return a type. What *is* the return type of a constructor?
4. How would you rewrite the `SayHello(String[] sAr)` constructor to use an *enhanced for* loop?
5. True or False: a constructor will always have the same name as the .java file in which it occurs.
6. When you call

```
new SayHello("");
```

(with nothing between the two double quotes), which one of the two constructors gets called initially: `SayHello()`, or `SayHello(String s)`?

If the call is then changed to

```
new SayHello(null);
```

which constructor gets called?

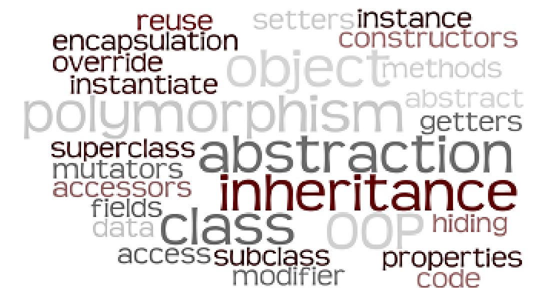
ALGONQUIN COLLEGE

Questions

7. In the three chained `SayHello()` constructors, if the third constructor is made private (as was suggested above),

```
public SayHello(){…}
public SayHello(String s){…}
private SayHello(String[] sAr){…}
```

which of the terms in the word cloud below right best describes the OOP philosophy behind this strategy?



ALGONQUIN COLLEGE

2.4 Importing Classes

Say we now wish to modify our earlier `SayHello` program so that it prompts the user for their name. For this, we require the `Scanner` class. The code looks like this:

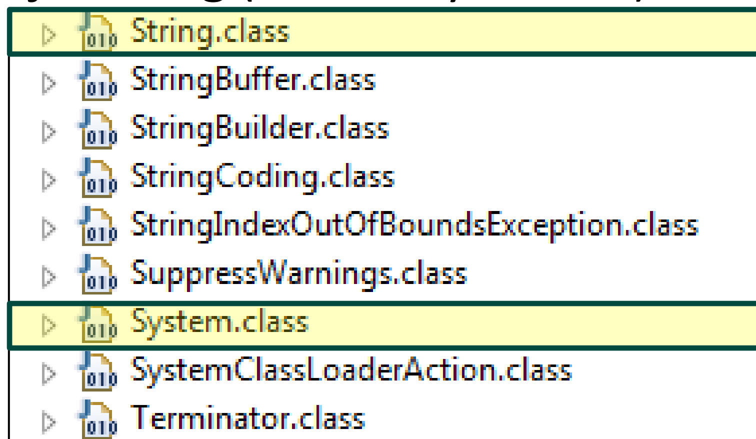
```
package cst8284.sayHello;
import java.util.Scanner;

public class SayHelloTest {
   public static void main(String[] args) {
      System.out.println("Please enter your name");
      Scanner name = new Scanner(System.in);
      new SayHello(name.inputNext());
   }
}
```

Liang 2.2 D&D 2.5

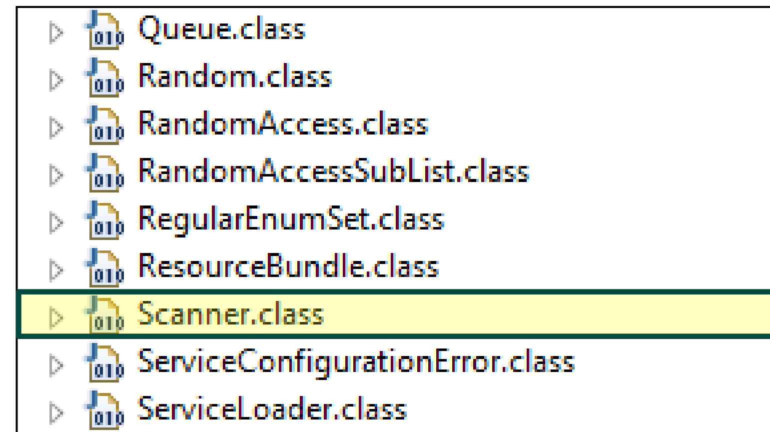
2.4 Importing Classes

Note the `import` statement. Scanner uses `java.util`, and while `java.lang` is imported by default, the `java.util` package is *not*. To access the classes in this package, including `Scanner`, you must import the package by name into your program using `import java.util.Scanner;`

java.lang (loaded by default):



java.util (must be imported):



2.4 Importing Classes

Note that there's nothing wrong with importing `java.lang.System`:

```
package cst8284.sayHello;
import java.lang.System;
import java.util.Scanner;
public class SayHelloHello {
   public static void main(String[] args) {
      System.out.println("Please enter your name");
      Scanner name = new Scanner(System.in);
        new SayHello(name.inputNext());
   }
}
```

While this won't trigger an error, it won't add anything that isn't already present. But it does serve to make the `java.lang` dependence of `System` obvious.

ALGONQUIN COLLEGE

2.4 Importing Classes

What does `import` do? Contrary to expectations, it *does not* import the `Scanner` code into your Java project for inclusion in the compiled product. The line

```
import java.util.Scanner;
```

tells the compiler: the `Scanner` class is to be found in the `java.util` package. If this line is not present, the connection between `Scanner` and `java.util` would be missing, and an error would be flagged. But you *can* do this instead:

```
java.util.Scanner name = new java.util.Scanner(System.in);
```

This works, but it is rather too much to write each time we use a new class. So the line

```
import java.util.Scanner;
```

is a shortcut for programmers; it does not actually import any code into your program. But it does allow for a connection to be made between a class (`Scanner`) and its location in memory (`java.util`) so that when the code *is* compiled, the location of that class's bytecode is already known.

2.4 Importing Classes

The wildcard symbol, *, can be used to import everything in a particular library, useful if you wish to import a number of classes from the same library. Thus

```
import java.lang.*;
```

says: import all the classes in this package. Again, this is a shortcut for programmers, merely designed to help tighten the code. Note that:

1. The '*' does not apply to packages in other libraries that begin with the same sequence. Thus the above command does not get you access to the classes in `java.lang.annotation`, `java.lang.instrument`, `java.lang.invoke`… or any of the other classes that begin with `java.lang`. This is important in Javafx, since most of the libraries we'll be using begin with `javafx.scene` or `com.sun.javafx`. But you cannot save yourself writing by simply importing `javafx.scene.*` and `com.sun.javafx.*`
2. As before, `import` only connects the classes in the library to the location of its code; it does not import any code into your program.

2.5 Objects

Before proceeding, we'll simplify the `SayHello` class somewhat and use only the no arg constructor in example that follows. We now assume:

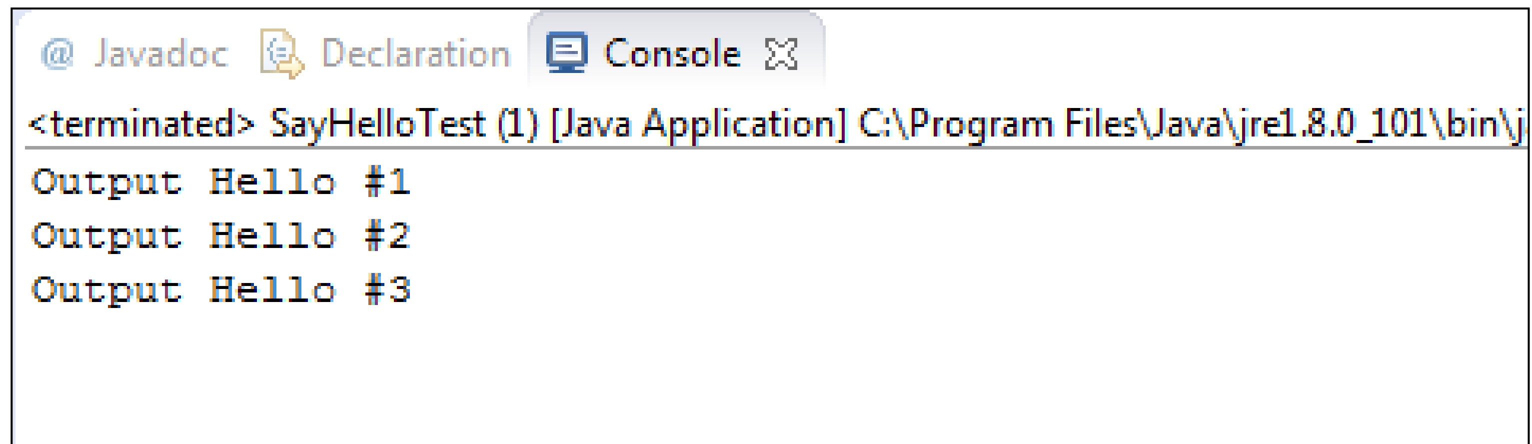
```
public class SayHelloTest {
   public static void main(String[] args){
      SayHello sh1 = new SayHello();
      SayHello sh2 = new SayHello();
      SayHello sh3 = new SayHello();
   }
}

public class SayHello {
   static int helloCtr = 0;  // static counter

   public SayHello(){        //constructor calls public method
      outputHello();
   }
   public void outputHello(){ //this method does all the work
      helloCtr++;
      System.out.println("Output Hello #" + helloCtr);
   }
}
```

2.5 Objects

When we run `SayHelloTest`, code execution again begins at `main()`, and executes through to the end. Inside `main()`, each time we invoke

```
SayHello shX = new SayHello();
```

we create a new instance of the `SayHello` object. This increments the `helloCtr` variable, and then prints out "`Output Hello #`", along with the new value.



Liang 9

D&D 8

2.6 `static` Members

Before we look deeper into this subject, an important distinction must be made between class members, and instance members, which we associate with instantiated objects:

1. Despite the fact that you load your Java classes whole, as complete entities, nothing requires that the compiled bytecode be stored in contiguous blocks of memory. Different processes are located at different locations according to the frequency of their use, their scope, and their size. (A more detailed discussion of this topic will be found in the video module on 'Stack v. Heap', to be presented later in the semester.)

2. Class, or **static**, members are loaded first and executed, while objects are instantiated and loaded into memory 'on the fly' as the program executes. Therefore, static members can only access other static members at the time the program begins execution, since instance members do not yet exist.

Liang 9.7 | D&D 8.11

2.6 `static` Members

So when our program begins execution, the class containing `public static void main()` is loaded into memory and that code begins execution.

This calls on the `SayHello` class, so the `SayHello` *class* gets loaded next. This will act as the template for any instantiated objects based on the `SayHello` class.

```
public class SayHelloTest {
  public static void main(String[] args){
    SayHello sh1 = new SayHello();
    SayHello sh2 = new SayHello();
    SayHello sh3 = new SayHello();
  }
}
```

```
public class SayHello {

   static int helloCtr = 0;

   public SayHello(){
     outputHello();
   }

   public void outputHello(){
   helloCtr++;
   System.out.println("Output Hello #"
     + helloCtr);
   }
 }
```

2.6 `static` Members

Finally, three instances of the `SayHello` object are loaded into memory, based upon the `SayHello` class.

```
public class SayHelloTest {
  public static void main(String[] args){
    SayHello sh1 = new SayHello();
    SayHello sh2 = new SayHello();
    SayHello sh3 = new SayHello();
  }
}
```

```
public class SayHello {
   static int helloCtr = 0;
   public SayHello(){
     outputHello();
   }

   public void outputHello(){
   helloCtr++;
   System.out.println("Output Hello #"
     + helloCtr);
   }
}
```

```
public class SayHello {
   …
   public SayHello(){
     outputHello();
   }

   public void outputHello(){
   helloCtr++;
   System.out.println("Output Hello #"
     + helloCtr);
   }
}
```

2.6 `static` Members

So `SayHelloTest` causes space to be reserved in memory for three *separate* instances of `SayHello`. But when using `helloCtr`, each instance refers to the same place in memory that was first associated with that static variable.

```
public class SayHelloTest {
  public static void main(String[] args){
    SayHello sh1 = new SayHello();
    SayHello sh2 = new SayHello();
    SayHello sh3 = new SayHello();
  }
}
```

sh1 **sh2** **sh3**

```
public class SayHello {
  static int helloCtr = 0;

  public SayHello(){
    outputHello();
  }

  public void outputHello(){
  helloCtr++;
  System.out.println("Output
       Hello #" + helloCtr);
  }
}
```

```
public class SayHello {
  static int helloCtr = 0;

  public SayHello(){
    outputHello();
  }

  public void outputHello(){
  helloCtr++;
  System.out.println("Output
       Hello #" + helloCtr);
  }
}
```

```
public class SayHello {
  static int helloCtr = 0;

  public SayHello(){
    outputHello();
  }

  public void outputHello(){
  helloCtr++;
  System.out.println("Output
       Hello #" + helloCtr);
  }
}
```

2.6 `static` Members

Or think of it this way: the gingerbread man cookie cutter is used as the template to instantiate new gingerbread man objects based on its shape. But certain components of the gingerbread man are referred back to the actual cookie cutter itself.

2.6 `static` Members

To repeat:

Members declared `static`

are loaded only once,

as part of the class, not the object.

Hence any reference to the static variable `helloCtr` in the `SayHello` object always refers to the same location *in the class the instance was derived from*.

2.6 `static` Members

Say we instantiate *one* instance of `SayHello` and call on the `outputHello()` method twice more, like this:

```
public class SayHelloTest {
    public static void main(String[] args){
        SayHello sh = new SayHello();
        sh.outputHello();
        sh.outputHello();
    }
}
```

2.6 `static` Members

The overall effect is the same as before: the message is output to the console three times. Even though the `SayHello` class is instantiated only once, the static property `helloCtr` is still with the class, not with the instance: static memory is not associated with any *instance* of a class, only with the class itself.

```
public class SayHello {

   static int helloCtr = 0;

   public SayHello(){
     outputHello();
   }

   public void outputHello(){
     helloCtr++;
     System.out.println("Output Hello #"
      + helloCtr);
   }
}
```

```
public class SayHello {

   public SayHello(){
     outputHello();
   }

   public void outputHello(){
     helloCtr++;
     System.out.println("Output Hello #"
     + helloCtr);
   }
}
```

So anything labelled `static` is loaded only once in memory, during startup; static members know nothing of the instances of classes

2.6 `static` Members

This applies even if we put *all* the code into `SayHelloTest` and have it instantiate separate instances of *itself*. The principle is exactly the same:

```
public class SayHelloTest{

    static int helloCtr = 0;

    public static void main(String[] args){
        SayHelloTest shTest1 = new SayHelloTest();
        SayHelloTest shTest2 = new SayHelloTest();
        SayHelloTest shTest3 = new SayHelloTest();
    }

    public SayHelloTest(){outputHello());

    public void outputHello(){
        helloCtr++;
        System.out.println("Output Hello #" + helloCtr);
    }
}
```

ALGONQUIN COLLEGE

2.6 `static` Members

Note that since `main()` is declared `static`, it too appears only once in memory: in the class, rather than in any of the objects instantiated by it.

shTest1 **shTest2** **shTest3**

```
public class SayHelloTest{

    static int helloCtr = 0;

    public static void main(String[] args){
        SayHelloTest shTest1 = new SayHelloTest();
        SayHelloTest shTest2 = new SayHelloTest();
        SayHelloTest shTest3 = new SayHelloTest();
    }

    public SayHelloTest(){
        outputHello();
    }

    public void outputHello(){
        helloCtr++;
        System.out.println(
            "Output Hello #" +
            helloCtr);
    }
}
```

2.6 `static` Members

In summary, there are four important things to note about static/class members:

1. Anything declared `static` is loaded only once in memory, during startup. Each instantiated object may contain static members, but this doesn't change the fact that each static member in an object exists only once in memory associated with the class, not the object itself.

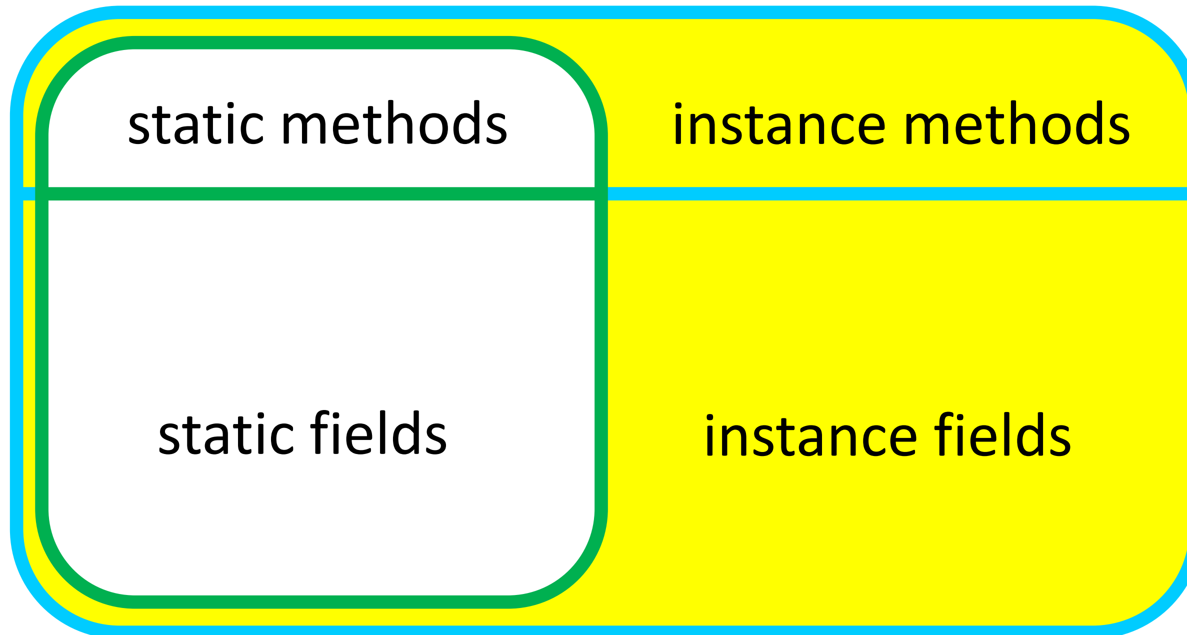
2. Because they are loaded first, static members know nothing about instances of classes, or anything OOP-related. While a static method can declare a local variable and set it equal to a new instance of an object, static methods are *never* tied to the instance of any class.

3. Because instance members are loaded *after* static members, instance members can reference static members. But a static member, being loaded first, cannot reference an instance member, of which it knows nothing.

ALGONQUIN COLLEGE

2.6 `static` Members

3. (con’t) So the relationship between instance members and class, or static, members is described by the following diagram:



Static methods talk to static fields; instance methods, being loaded later, can talk to instances fields *and* static members as well.

ALGONQUIN COLLEGE

2.6 `static` Members

4. Any member not explicitly declared `static` is automatically an instance member: it must be instantiated in an object before it can occupy space in memory and be executable. The exception to this rule are the local variables declared *inside* static methods, which are declared and loaded with the class itself. They may be regarded *as if* they were a static part of the static method (although internally, they are treated differently by the JVM than truly static fields).

2.6 `static` Members

Q. Of what use are static members?

A. We've already seen one typical use: `helloCtr` keeps track of the number of instances of an object. If a new object is created and it needs to know how many `SayHello` objects exist, it merely needs to consult the value of `helloCtr`.

Q. How do you access a static member in another class?

A. To access a static field from outside the class in which it is declared, use the form:

```
ClassName.staticFieldIdentifier
```

Thus to access `SayHelloTest`'s `helloCtr` static variable from another class (using the pre-modified version of the `SayHelloTest` program from three slides back), you would write:

```
SayHelloTest.helloCtr
```

2.6 `static` Members

Of course, it's not a good idea to access an object's members directly—static or otherwise—so you should provide static getter and setter methods to access the counter itself. Let's create a class designed to keep track of the static counter, and supply it with appropriate methods:

```
public class Counter {
    private static int ctr = 0;

    public static int getCounter(){return ctr;}
    public static void resetCounter(){ctr = 0;}
    public static void iterateCounter(){ctr++;}

}
```

Note that the counter value, `ctr`, is made `private`. This ensures that it is only visible inside the class, and can only be accessed via the public methods provided.

2.6 `static` Members

We can use these static methods as follows.

```
public class SayHelloTest {
   public static void main(String[] args) {
      for (int i = 0; i < 3; i++)
         new SayHello();
     }
}

public class SayHello {
   public SayHello(){
         Counter.iterateCounter();
         outputHello();
   }

   public void outputHello() {
      System.out.println("Output Hello #"
         + Counter.getCounter());
   }
}
```

2.6 `static` Members

Static methods are also useful when the same result will be produced regardless of the number of objects. For example, the following class, `OrdinalSuffix`, returns a two-character string that corresponds to the appropriate suffix associated with each number: 1**st**, 2**nd** , 3**rd**...119**th** , 120**th**, 121**st**...etc.

```
public class OrdinalSuffix {
   public static String getSuffix(int n){
      int m = n%100;
      if ((m==10) || (m==12) || (m==13))
         return ("th");
      else {
         switch (n%10){
            case 1: return "st";
            case 2: return "nd";
            case 3: return "rd";
            default: return "th";
         }
      }
   }
}
```

ALGONQUIN COLLEGE

2.6 `static` Members

Now if each new `sayHello` object uses the code…

```
public class SayHello {

    public SayHello(){
        Counter.iterateCounter();
        outputHello();
    }

    public void outputHello(){
        int currentCtr = Counter.getCounter();
        System.out.println("Hello the " + currentCtr +
            OrdinalSuffix.getSuffix(currentCtr)
                + " time.");
    }
}
```

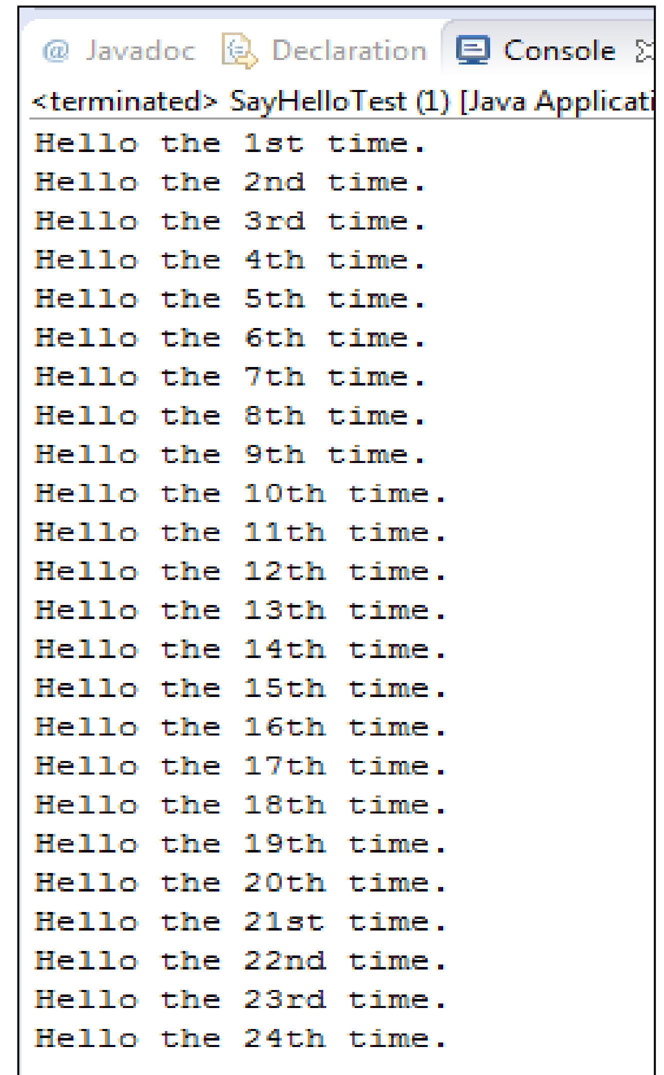
ALGONQUIN COLLEGE

2.6 `static` Members

…and if we run the following `SayHelloTest`,

```
public class SayHelloTest {
  public static void main(…){
    for (int i=0; i < 24; i++)
      new SayHello();
  }
}
```

…then the output is shown at right. Note that the string suffix (`st`, `nd`, `rd`, or `th`) associated with each number is correct.



2.6 `static` Members

The method `getSuffix` should be `static` because it produces the same results regardless of the object in which it is used; the instance of the object is irrelevant, and only the integer returned by the `getCounter()` method matters.

So methods should be declared `static` when their operation is independent of the instance in which they're used.

See: http://stackoverflow.com/questions/2671496/java-when-to-use-static-methods for some very detailed explanations of when to use static.

2.6 `static` Members

Think of `static` as a memory management device. If a segment of code does not need to be instance dependent, then it only *needs* to be loaded once…then it should be `static`. This saves memory and increases the loading speed of your programs. It is especially important when:

1. You're loading thousands or millions of instances of an object, and both loading speed and memory space are concerns;
2. You're programming an embedded device, i.e. one with limited memory, and every byte counts.

And as we've seen, it is also *safer* for programs to make certain members static, to restrict their access to client objects to a single point of contact.

ALGONQUIN COLLEGE

2.6 `static` Members

Static members are used often in Java. For example, all of the properties and methods of the `Math` library are static. These include:

`Math.sin(radians)`	`Math.exp(x)`	`Math.ceil(x)`	`Math.PI`
`Math.cos(radians)`	`Math.log(x)`	`Math.floor(x)`	`Math.E`
`Math.tan(radians)`	`Math.log10(x)`	`Math.rint(x)`	
`Math.toRadians(degree)`	`Math.pow(x,y)`	`Math.round(x)`	
`Math.toDegrees(radians)`	`Math.sqrt(x)`	`Math.min(x,y)`	
`Math.asin(x)`	`Math.random()`	`Math.max(x,y)`	
`Math.acos(x)`	`Math.atan(x)`	`Math.abs(x)`	

Why static? Math operations never depend on any particular instance; π is always 3.1415… regardless of the instance in which it is used.

Note: The Math Library is one of the libraries that is downloaded automatically with `java.lang`. *See:* *http://www.tutorialspoint.com/java/lang/java_lang_math.htm* for details

D&D 6.3

2.6 `static` Members

Note that Java lacks static classes (although they appear, for example, in C#). If they existed, the `Math` class would be a good candidate to be made `static`.

Java does allow for **static initializer**s. For example, you may see, inside a class, the following:

```
static {

   // class-level initialization happens here

}
```

Code inside the block is loaded once into memory when the class loads. This can be useful for initializing static members *before* `main()` executes.

2.6 `static` Members

So you should use `static` when a member is independent of the instance of any object.

For example, a class `Car` might have getters and setters for speed, oil pressure, gas, temperature, mileage, colour, transmission, etc. These are obviously *not* static methods, since they specific features of each particular `Car`.

However, the factory that made the car is independent of the car itself. (The car may have a make and model, but again these are features of the car which can be stored as strings during instantiation via a constructor.)

2.6 `static` Members

On this note: the relationship between a vehicle and its features—its colour, speed, etc.—is different than the relationship between the vehicle and the factory that made it. For example, a truck has certain features that characterize it. The truck has a steering wheel, tires, a gas tank, etc. The relationship is one of **composition**.

Instance variables are well-suited for a compositional relationships. On the other hand, The factory that made the truck has a loose **association** with the vehicles it produces. But at no point can it be said that a truck is a *property* of the factory.

Static methods are suitable for producing objects; but by their very nature they are not suitable for describing the properties *of* an object. The relationships *between* objects is a subject that will be explored more fully in the next module.

Questions

1. What feature of `System.out` and `Math.PI` indicates that these are static members, and not instance members?

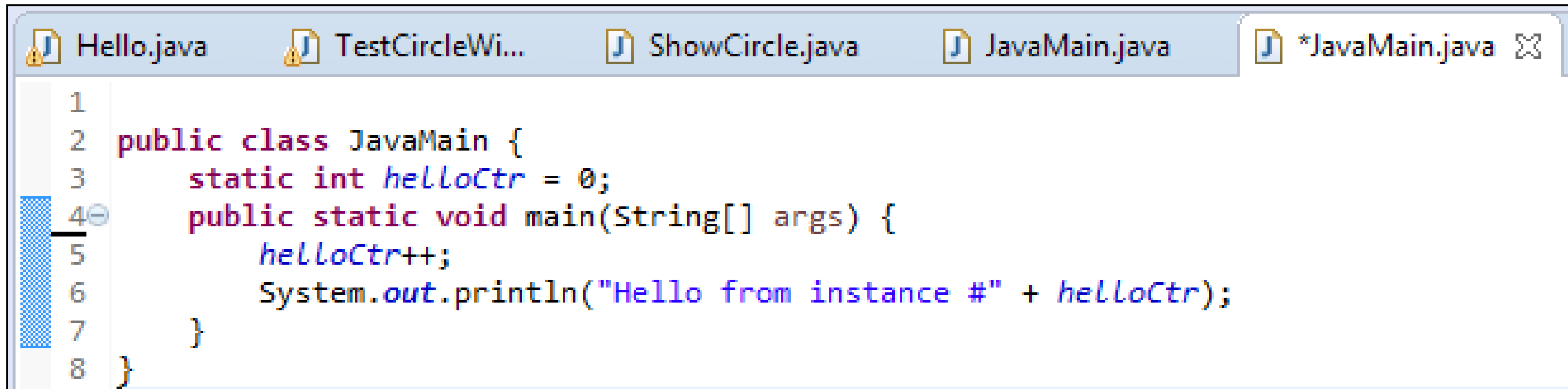
2. According to the Java documentation, the `in` property (of `System.in`) is of type `InputStream`; this tells you something about what the declaration of `in` must look like. Write out the probable form of this declaration based entirely on the information provided in the documentation.

3. Constructors cannot be declared `static`. Can you suggest a reason why this should be the case?

Questions

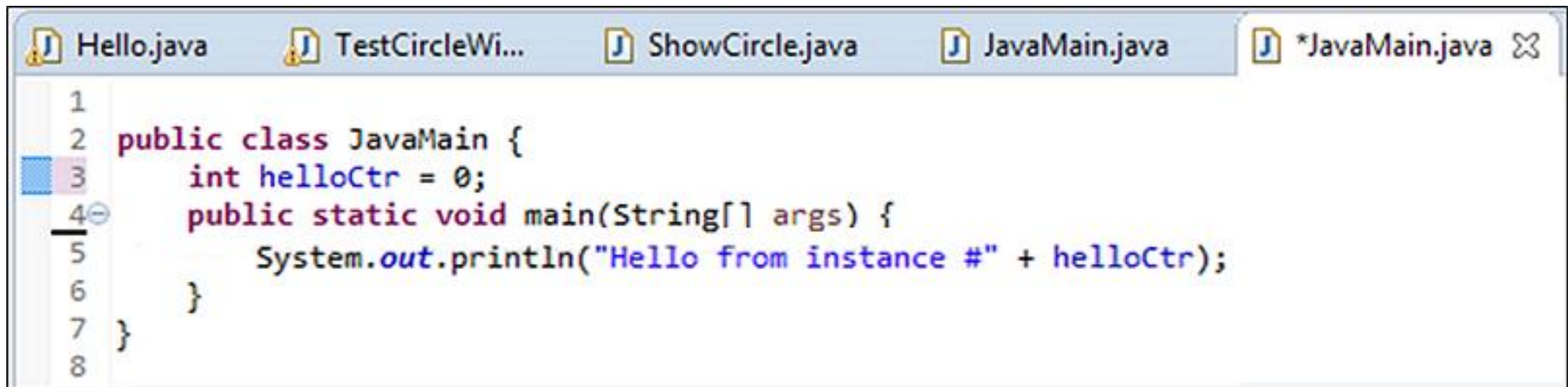
4. For each of the following code fragments, determine whether or not an error will be generated, according to the rules regarding access to static members

(a)




```
1
2 public class JavaMain {
3     static int helloCtr = 0;
4     public static void main(String[] args) {
5         helloCtr++;
6         System.out.println("Hello from instance #" + helloCtr);
7     }
8 }
```

(b)




```
1
2 public class JavaMain {
3     int helloCtr = 0;
4     public static void main(String[] args) {
5         System.out.println("Hello from instance #" + helloCtr);
6     }
7 }
8
```

ALGONQUIN COLLEGE

Questions

(c)

Hello.java | TestCircleWi... | ShowCircle.java | JavaMain.java | *JavaMain.java

```
1
2  public class JavaMain {
3      static int helloCtr = 0;
4      public static void main(String[] args) {
5          sayHello();
6      }
7
8      private void sayHello(){
9          helloCtr++;
10         System.out.println("Hello from instance #" + helloCtr);
11     }
12 }
```

(d)

Hello.java | TestCircleWi... | ShowCircle.java | JavaMain.java | *JavaMain.java

```
1
2  public class JavaMain {
3      static int helloCtr = 0;
4      public static void main(String[] args) {
5          sayHello();
6      }
7
8      private static void sayHello(){
9          helloCtr++;
10         System.out.println("Hello from instance #" + helloCtr);
11     }
12 }
```

ALGONQUIN COLLEGE

Summary

Let's review the various features of one variant of the 'Hello World' program:

```
package cst8284.sayHello;
```

the package name determines the folder for both the .java source and .class compiled code

```
import java.util.Scanner;
```

`import` alerts the JVM to search for the code associated with the class name in .jar files

```
public class SayHelloTest{
```

a class is the container for properties and methods, collectively referred to as members

```
  public static void main (String[] args){
```

`main()` is a static method which serves as the point of entry for the program. All local variable declared inside `main()` are treated as static variables

```
    Scanner input = new Scanner(System.in);
```

Scanner's code is located in java.util; Scanner is an overloaded constructor

```
    System.out.println("Enter your name ");
```

The System class can instantiate an out object, which has a println() method

```
    new SayHello(input.nextLine());
  }
```

new objects are instantiated using the new keyword; the constructor invoked depends on the signature

ALGONQUIN COLLEGE

Summary

```
package cst8284.sayHello;
```

the package name determines where classes are located and helps organize your code. 'Package private' constitutes a fourth level of access modification

```
public class SayHello {
```

Constructors must have the same name as their class

```
    SayHello(){
```

A no-arg constructor should be supplied with every class

```
        this("World");
    }
```

`this` refers to the current class, hence this line indicates a chained reference to another constructor

```
    SayHello(String name){
```

Multiple overloaded constructors are possible, provide they each have a different signature

```
        System.out.println("Hello " + name);
    }
}
```