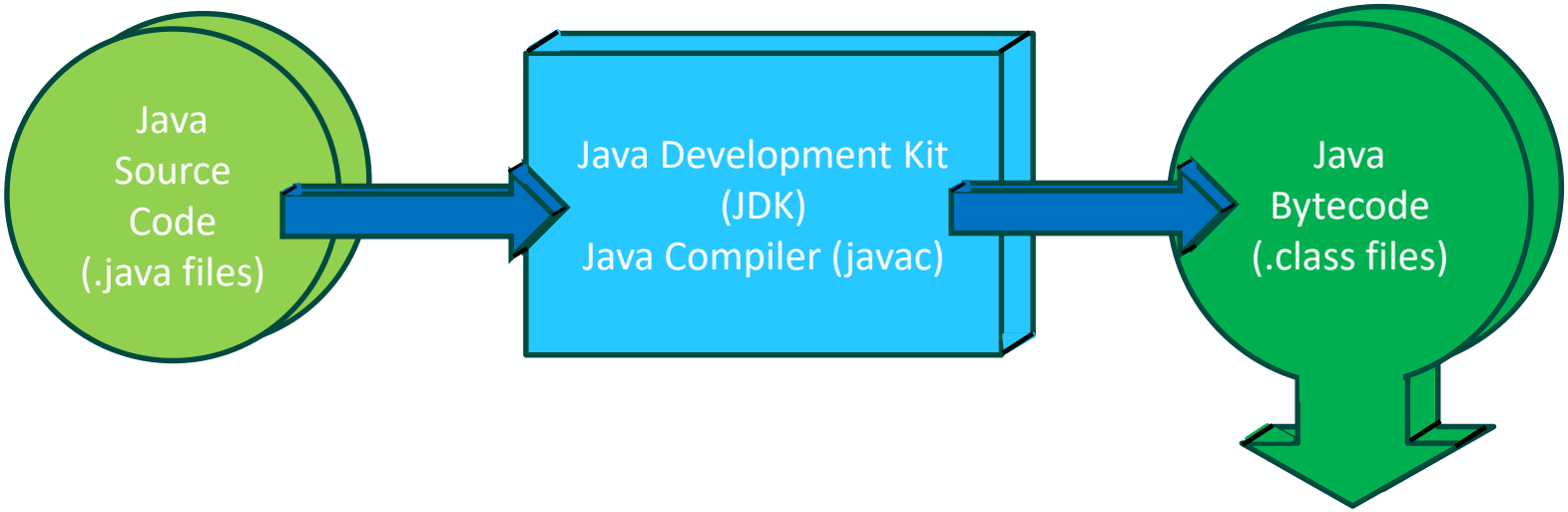
ALGONQUIN COLLEGE

MODULE 02: JAVA FOUNDATIONS I: COMPILATION

Professor : Dave Houtman
Office: T323
Office Hrs: Friday 11:30 – 12:45
Email: houtmad@algonquincollege.com

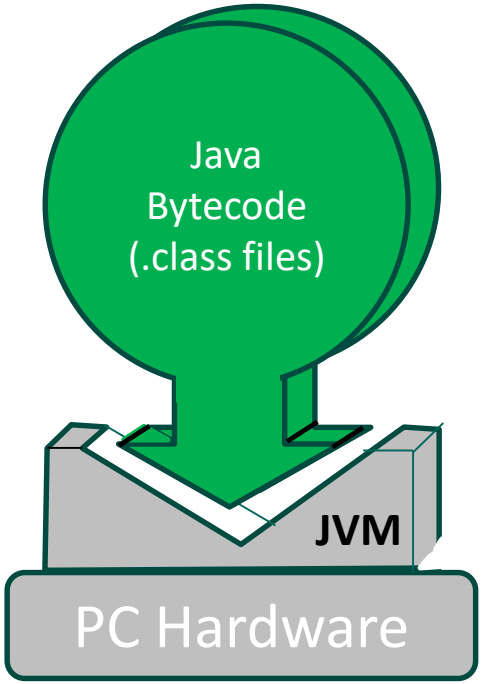
2.0 Runtime

Java code must first be converted to **bytecode** using the java compiler, known as `javac`. The bytecode is then stored in a **class file**.



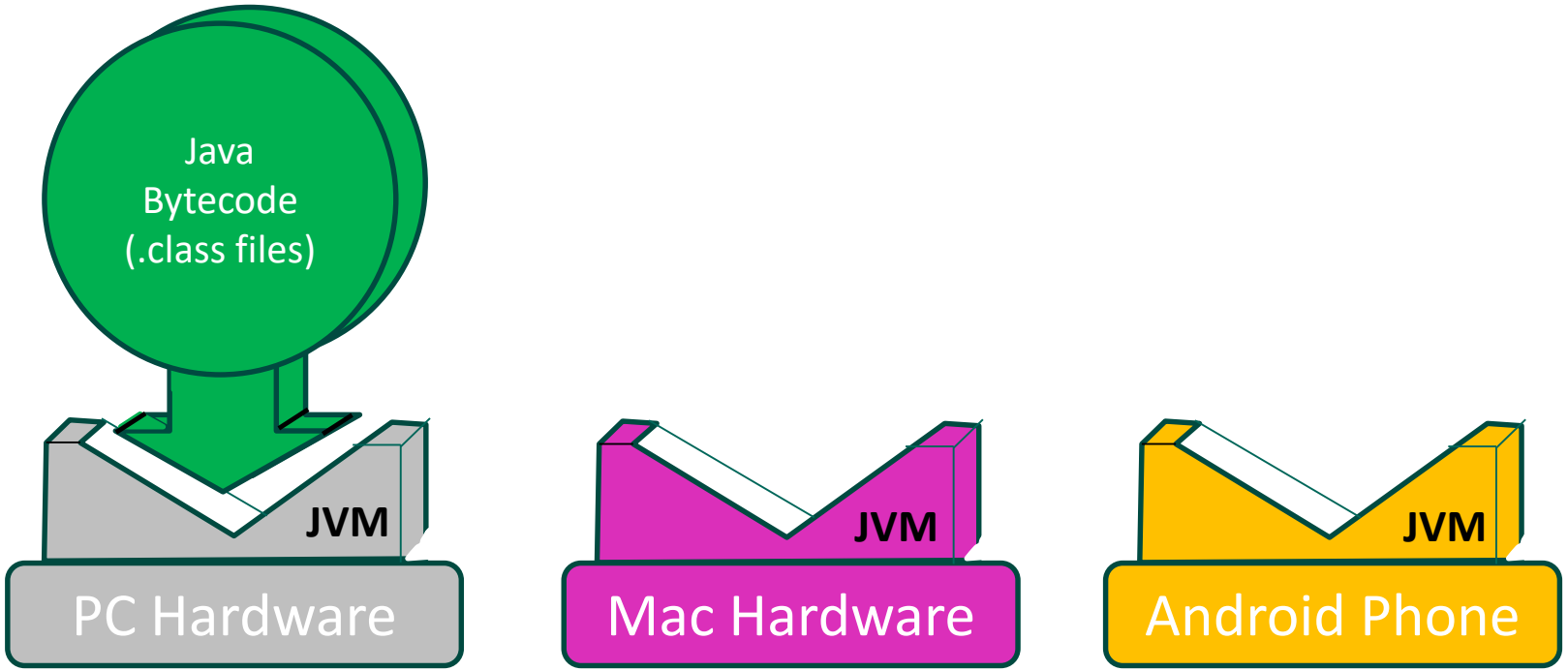
2.0 Introduction

This bytecode is then interpreted by the Java **interpreter**, called the **Java Virtual Machine** (**JVM**), which is a special piece of software loaded on your PC. The extra step of interpreting the bytecode (rather than compiling it down to executable code that can be run directly off the processor) comes at some cost in performance. For this reason, parts of the bytecode may be further compiled down to machine language for speed, and then executed directly off the processor.



2.0 Introduction

So Java is fundamentally an interpreted language—albeit one that needs to be compiled at different stages—with the JVM acting as a bytecode interpreter. Thus bytecode serves as a universal low-level language, allowing any java program to be executed on *any* piece of hardware that has a JVM loaded.



2.0 Introduction

The `main()` method is the starting point for program execution in any Java program (although, as we'll see, it is, strictly speaking, *not* required). '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 typical 'Hello World' program* is:

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

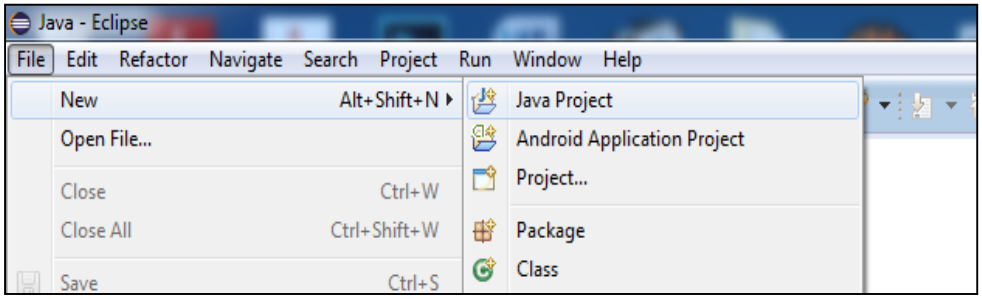
In Java, as in most programming languages, things are often not so simple as they on the surface: there's a great deal going on here. In this module we look at issues related to the compilation of code; in Module 03, we focus on run-time issues.

* For a list of 'Hello World' in ~500 other languages, including 'Human', see: https://helloworldcollection.github.io/

2.1 Workspace

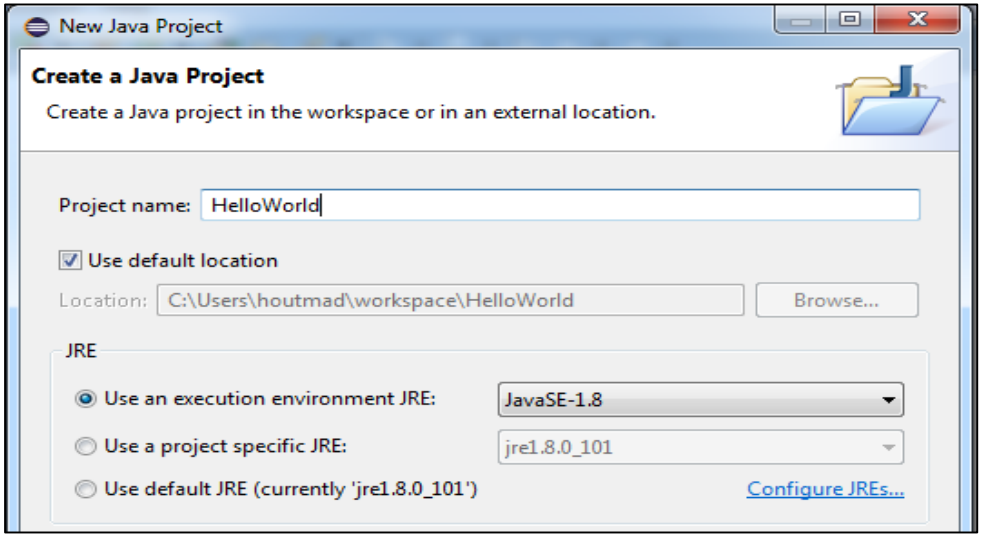
To build a new application, you first perform a series of steps in Eclipse which include:

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, as long as it is a valid identifier. 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 Workspace

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 sub-folders and files. This new folder is typically located in `C:\Users\your_username\eclipse-workspace`, seen below at right.

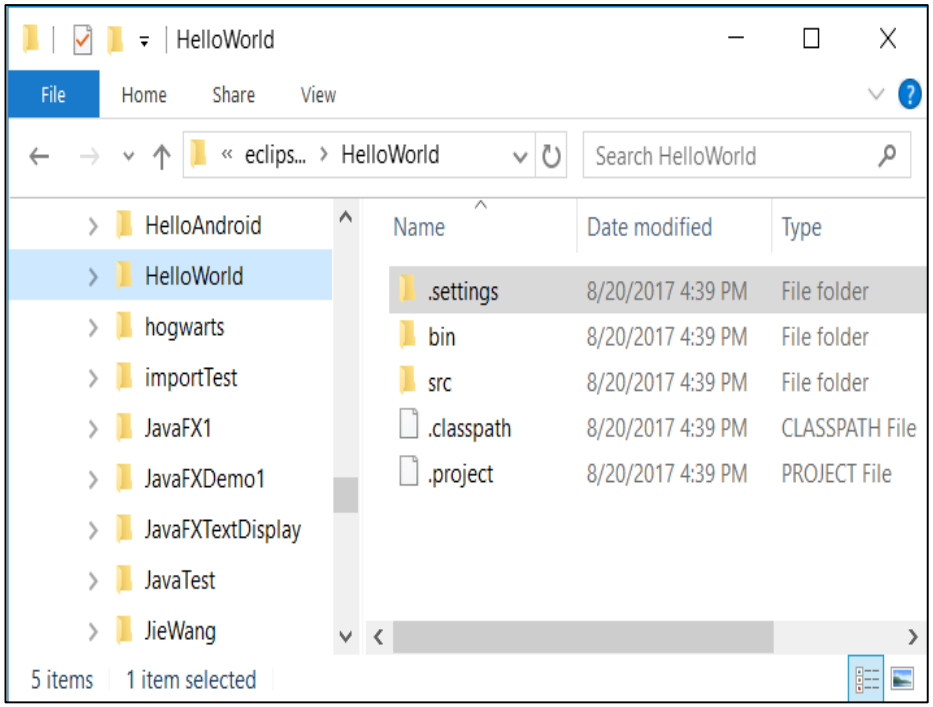
Each new project folder contains three subfolders and two files:

`src` – holds your .java code

`bin` – holds the .class bytecode for each compiled .java class in `src`

`.settings` – contains Eclipse general startup preferences

`.classpath` and **`.project`** are XML files that tell Eclipse how to find and build your project code



2.1 Workspace

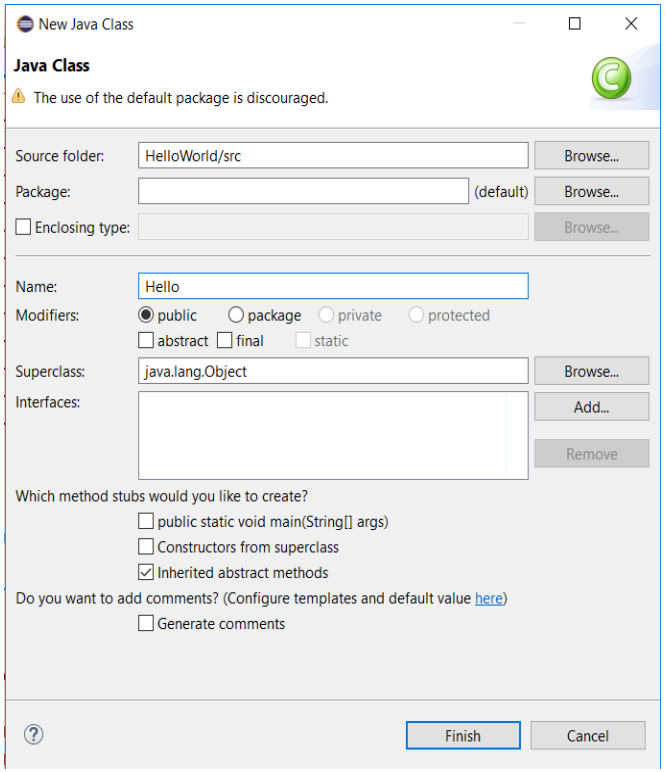
Continuing with our HelloWorld project: to add Java code to our Eclipse project, we first need to create a class file, as follows:

3) Select `File >> New > Class` from the menu;

4) Enter the class name in the `Name` textbox. We'll use `Hello` for our class name. Note that, by convention, class names are capitalized

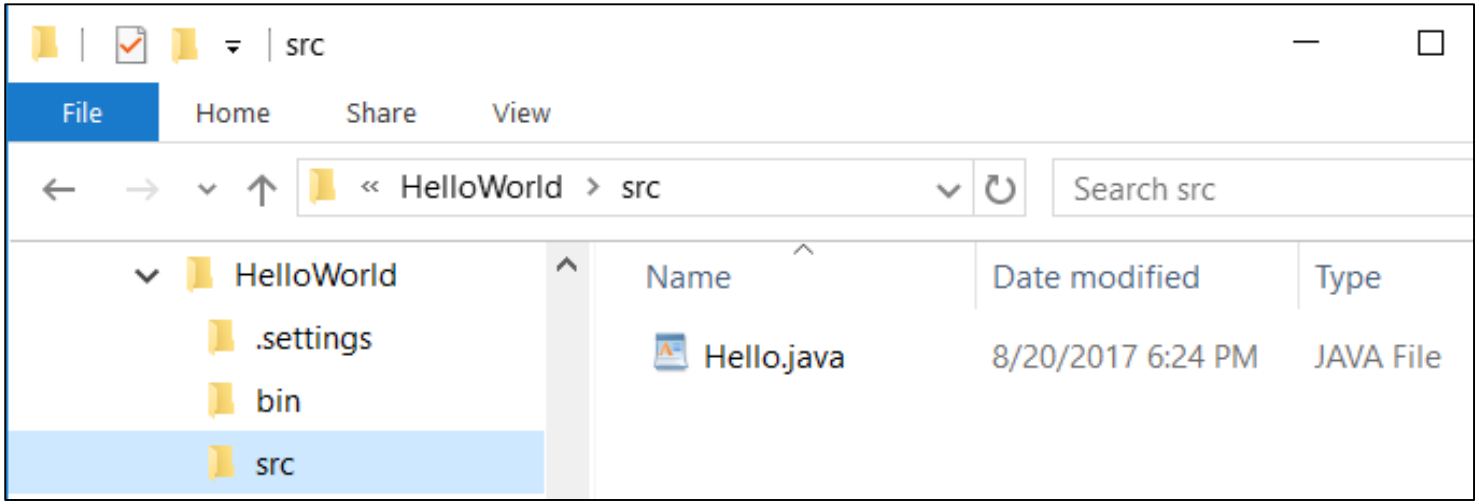
5) Click the `Finish` button

6) Enter the code into `main()`, like that shown in the `Hello` program provided earlier.



2.1 Workspace

In Windows, when we click `Finish`, the Hello.java file is loaded into the src directory of our project workspace:



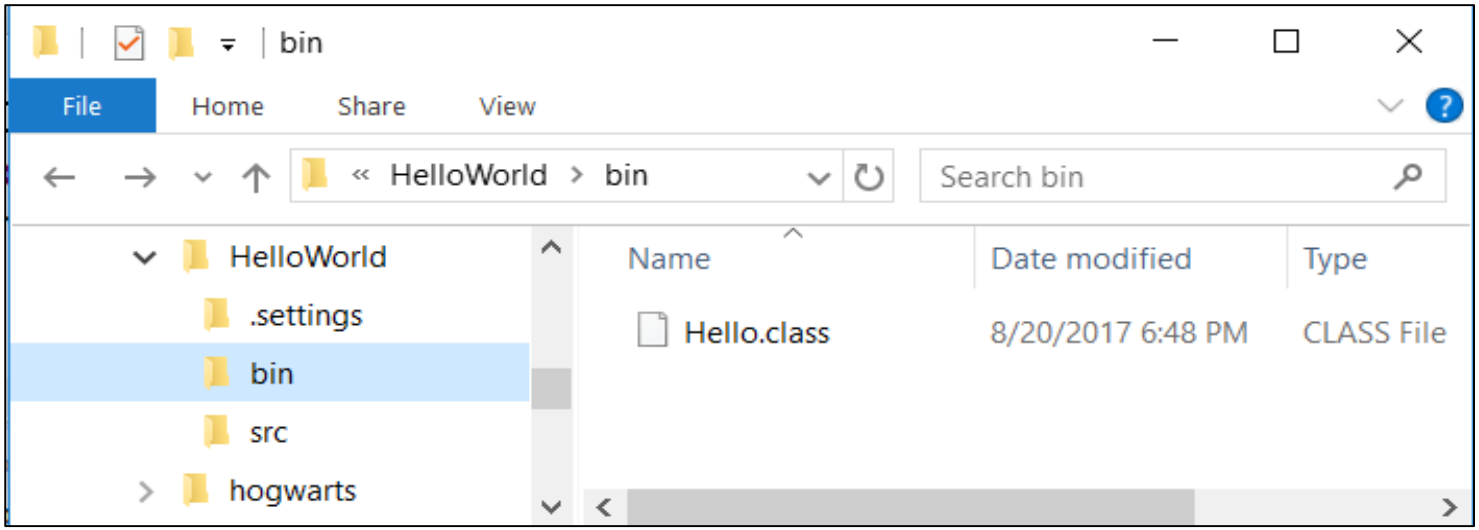
At this point, the file only contains the class declaration

```
public class Hello {

}
```

As we add Java code to our file and save it, that code will be saved to the java file in the `src` subfolder inside the HelloWorld workspace folder.

2.1 Workspace

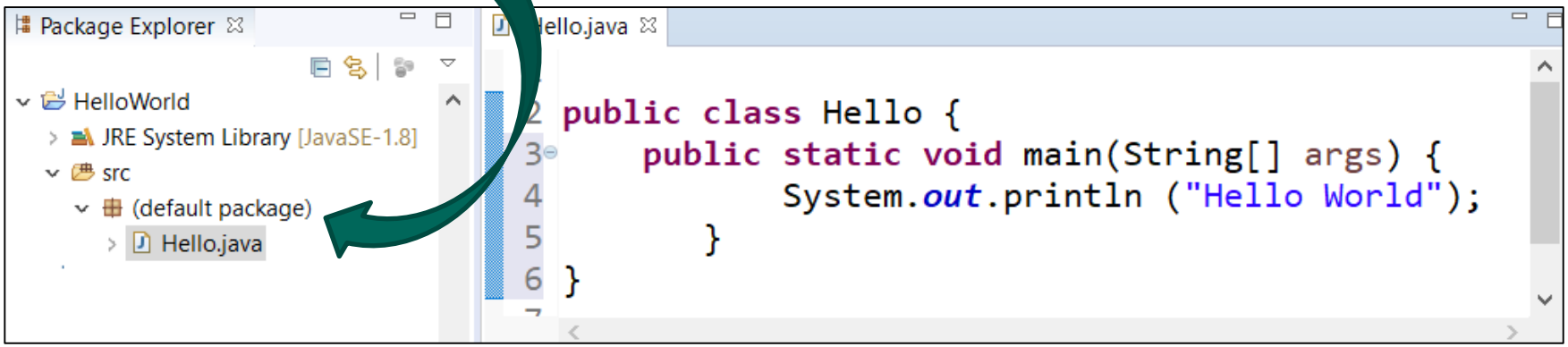
In Eclipse, when you first run Hello.java (or even if you just debug it), `javac` is called internally, and a .class file is produced. Class files are stored in the project's `bin` folder:



So running a program in Eclipse in any fashion has the effect of compiling java code in the `src` folder into bytecode, which is stored in the `bin` folder in a class file having the same name as the java file it originated from.

2.2 Packages

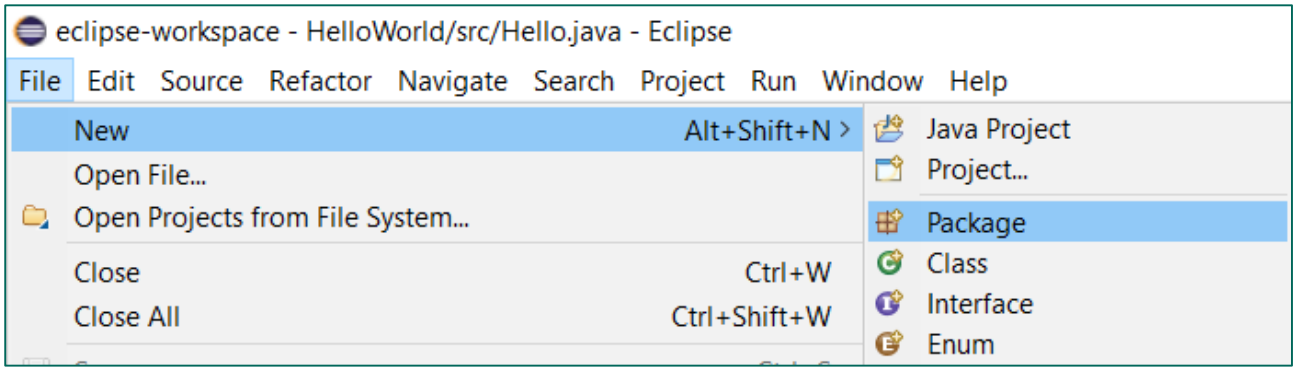
Consider again the setup in Eclipse. Notice that the `src` folder contains a `(default package)`.



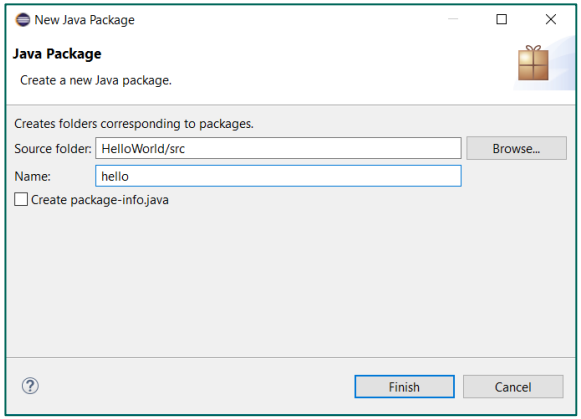
In general, we want avoid using the `default` package. All your code should be stored in named packages, which should ideally be added *after* the project has been created, but *before* the classes have been added; this saves you from moving .java files around afterward.

2.2 Packages

So after you've created your new project, with the project selected, click `File >> New >> Package` from the menu, or right-click on the project name, and select `New >> Package`



Then, in the dialog that appears, give your package an appropriate name:

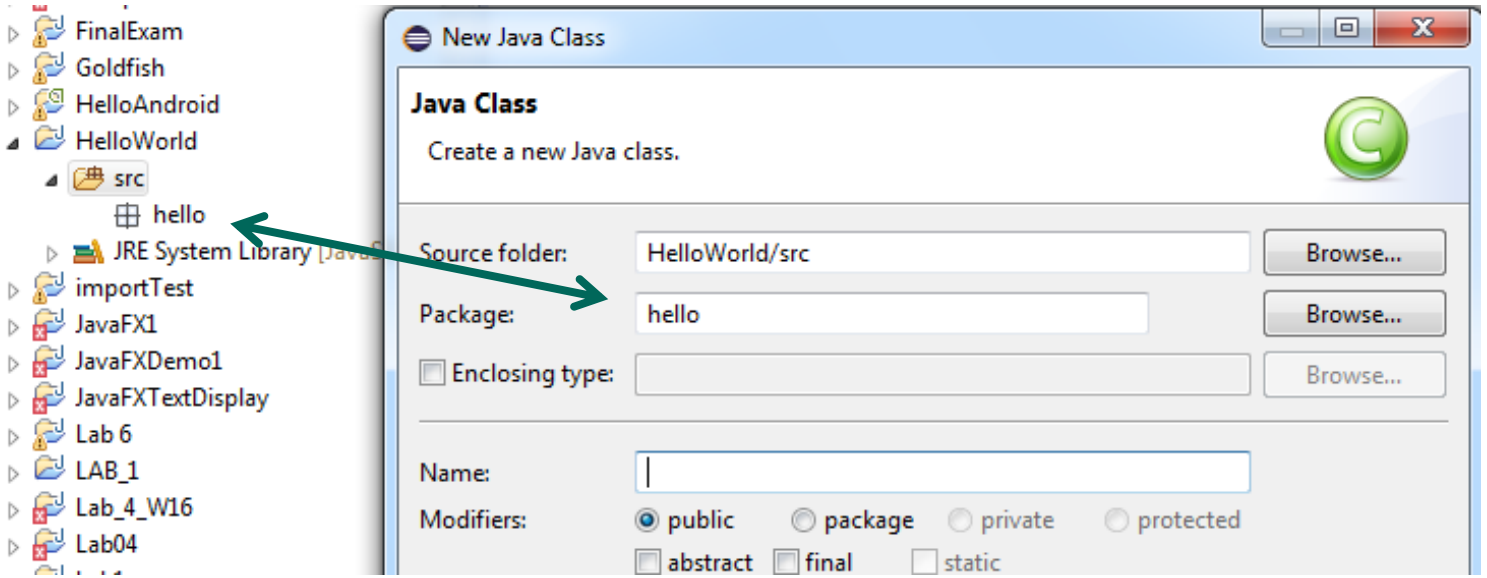


Liang 9.8 **D&D 8.14**

2.2 Packages

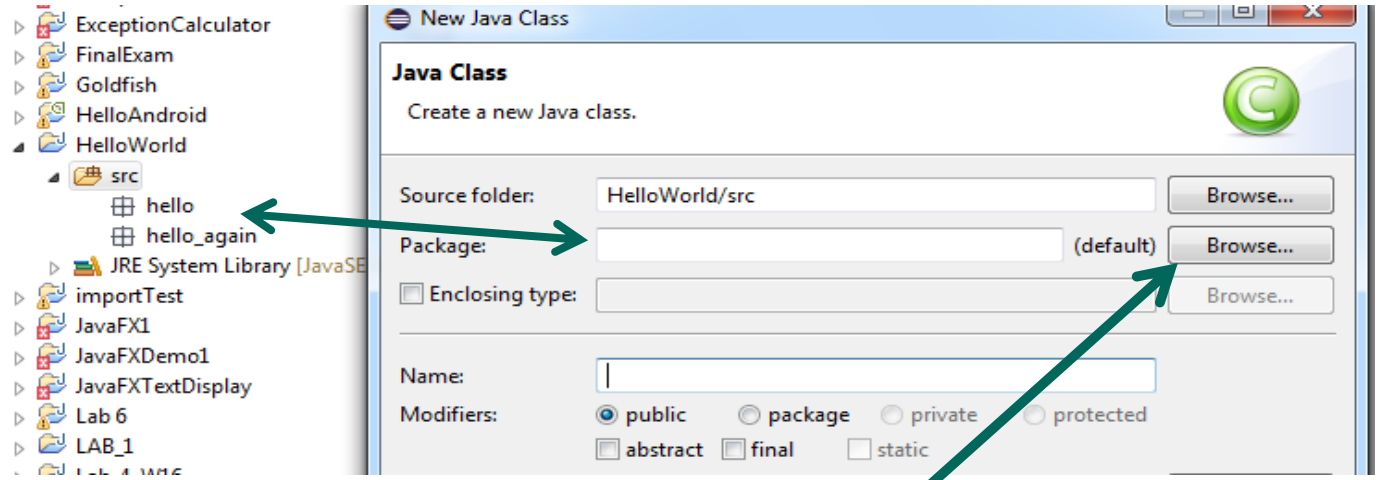
Once the package has been declared, Eclipse detects this automatically when you want to add classes to your project.

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

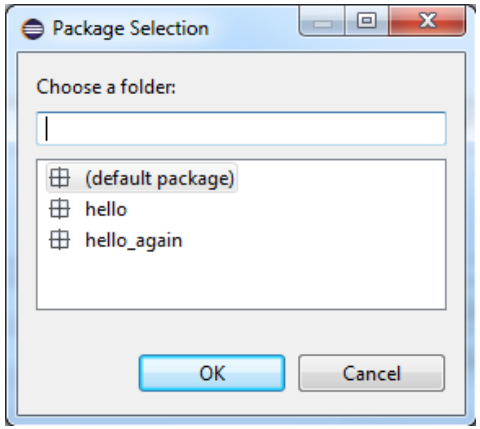


2.2 Packages

…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.

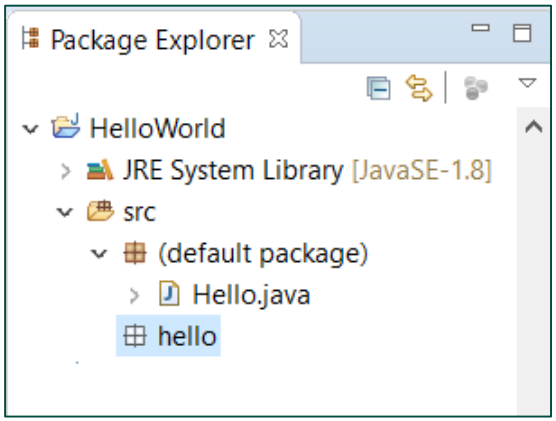


In this case, you should select the 'Browse' button and specify which of the packages the new class belongs in, otherwise your new class will be stored in the default package. Note that the default package is still there; it 'lives' in the `src` folder of your workspace (whether there's any .java files stored in `src` or not).

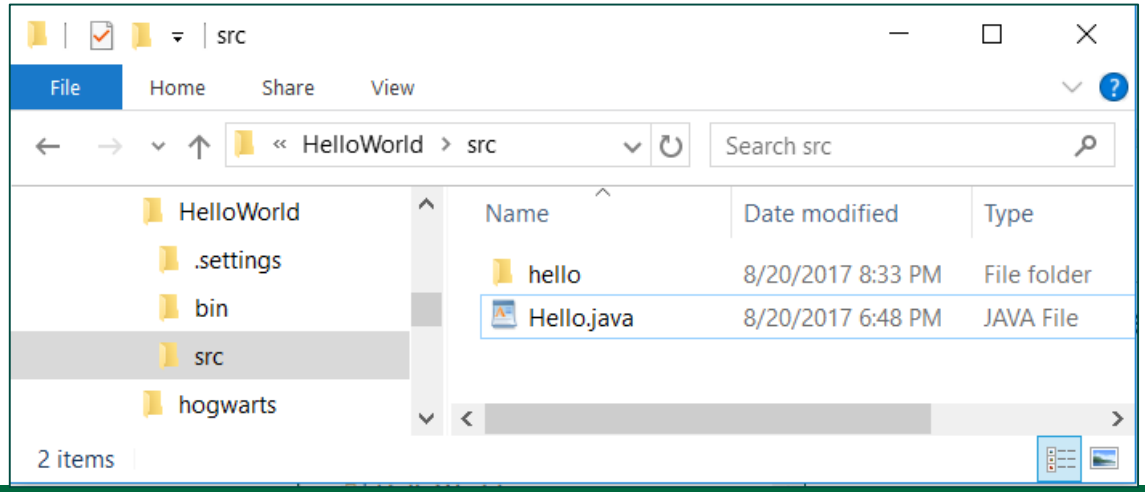


2.2 Packages

Say we add a package after we've already stored a java file in the default package. Once a package is created, it appears in Eclipse in the Package explorer…

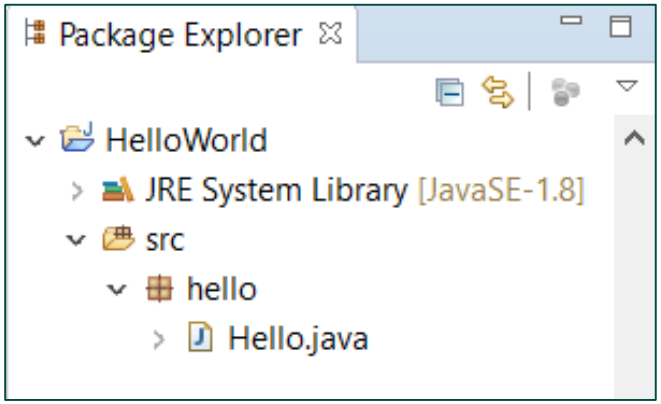


…and the package name becomes the name of a subfolder in `src`.

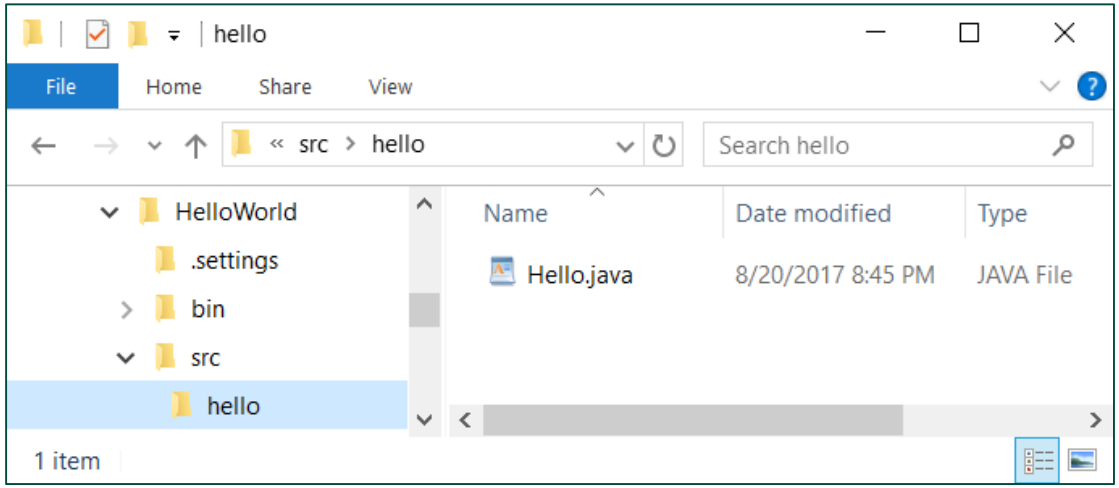


2.2 Packages

If we wish to relocate our java files to a package *after* the file has already been created, we can drag and drop that file directly into our package in Eclipse.



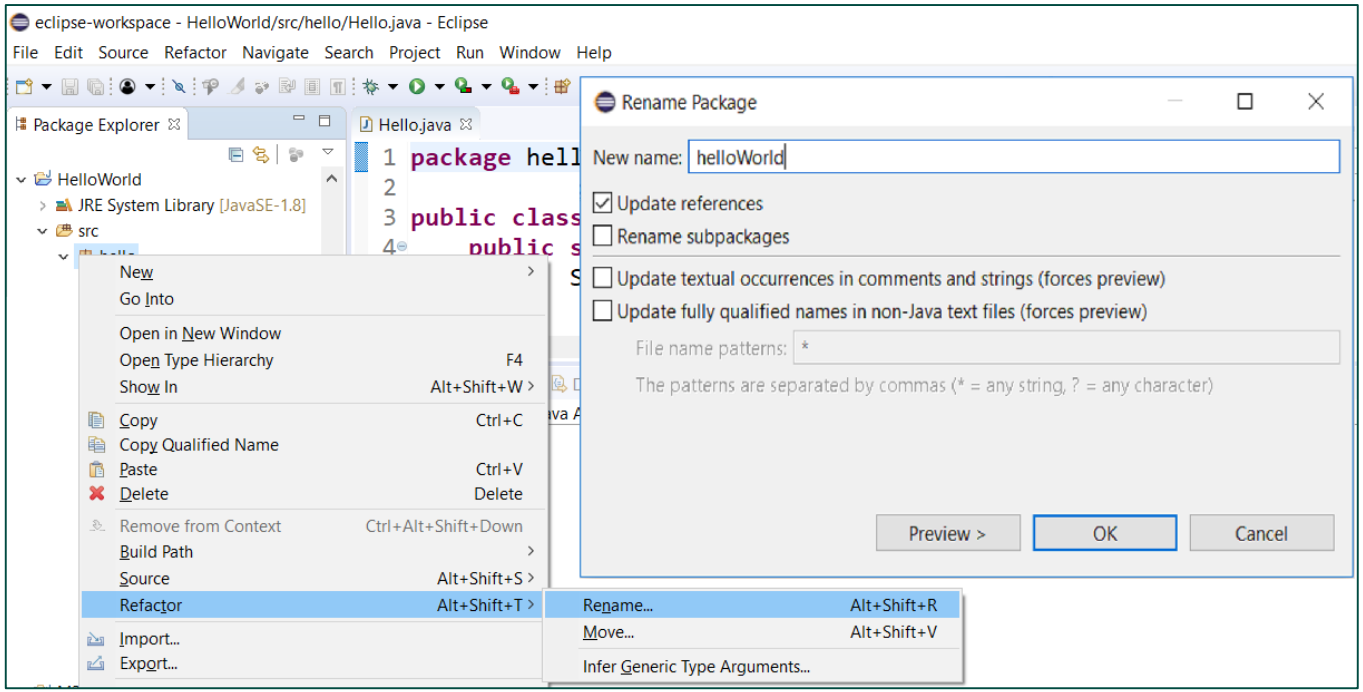
Of course, these changes are reflected in the underlying folders of the workspace:



Note that the default package disappears from Eclipse, since its folder is empty.

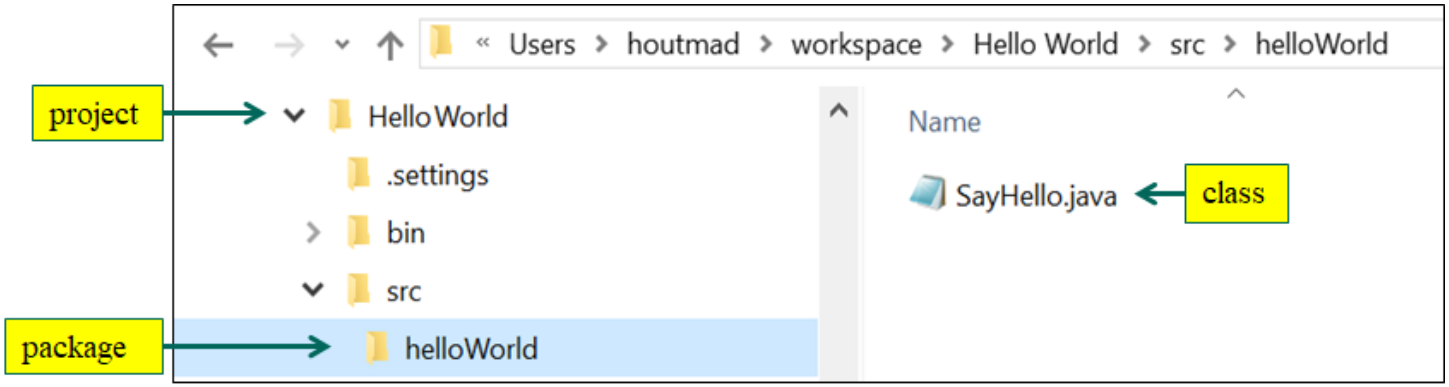
2.2 Packages

Finally, what if we wish to change the name of a class after it has been created? Copying the original Java code into a new class is inefficient and fraught with possible complications. A better way is to **refactor** the code using Eclipse, that is, *to change it internally while keeping its original behaviour intact*. You can change the name of a project, package, or class by right-clicking on it in Package Explorer, and then following the menu. In this way we can change the `hello` package to `helloWorld`.

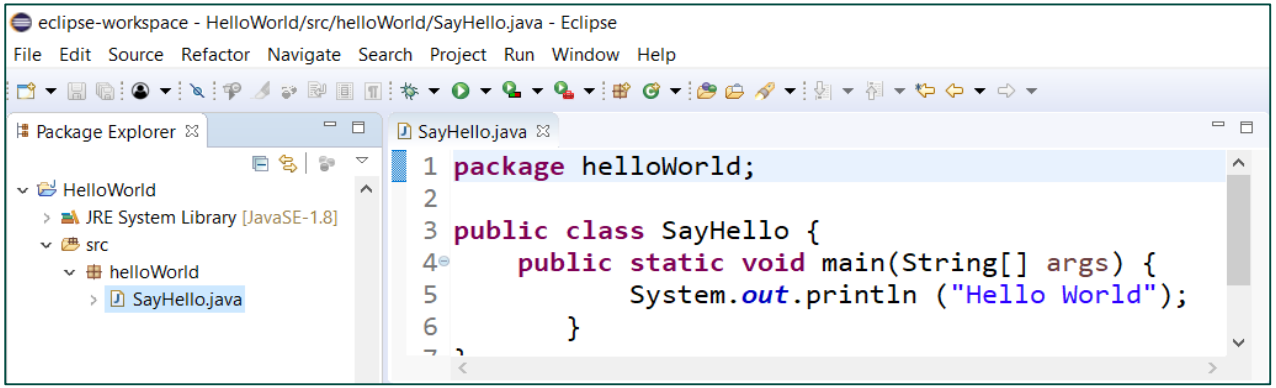


2.2 Packages

Similarly, we can refactor the Java file name from `Hello` to `SayHello`. As you'd expect, these changes are reflected in Windows Explorer.



Any references to the class name inside the file are updated to reflect the change. 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.



2.2 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) By convention, the name of packages starts with a lowercase character

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 the other .java files that share the same **namespace**. Thus each .java file in the `helloWorld` package must have

```
package helloWorld;
```

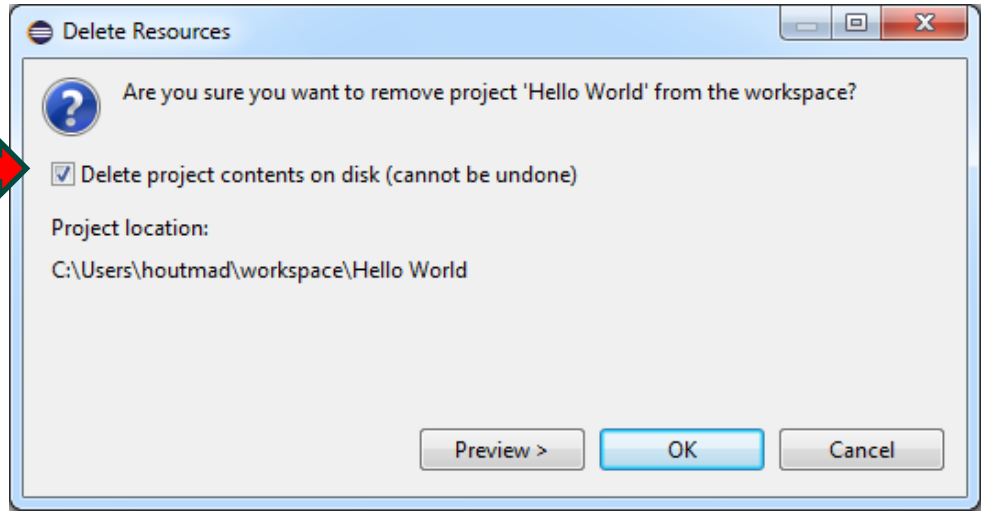
as the very first line. We'll discuss namespaces in Module 03. (Note: To keep our code terse in the examples that follow, we'll ignore this requirement)

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.

2.2 Packages - Notes

5) 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 project contents…' 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 will 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



2.3 JRE System Library

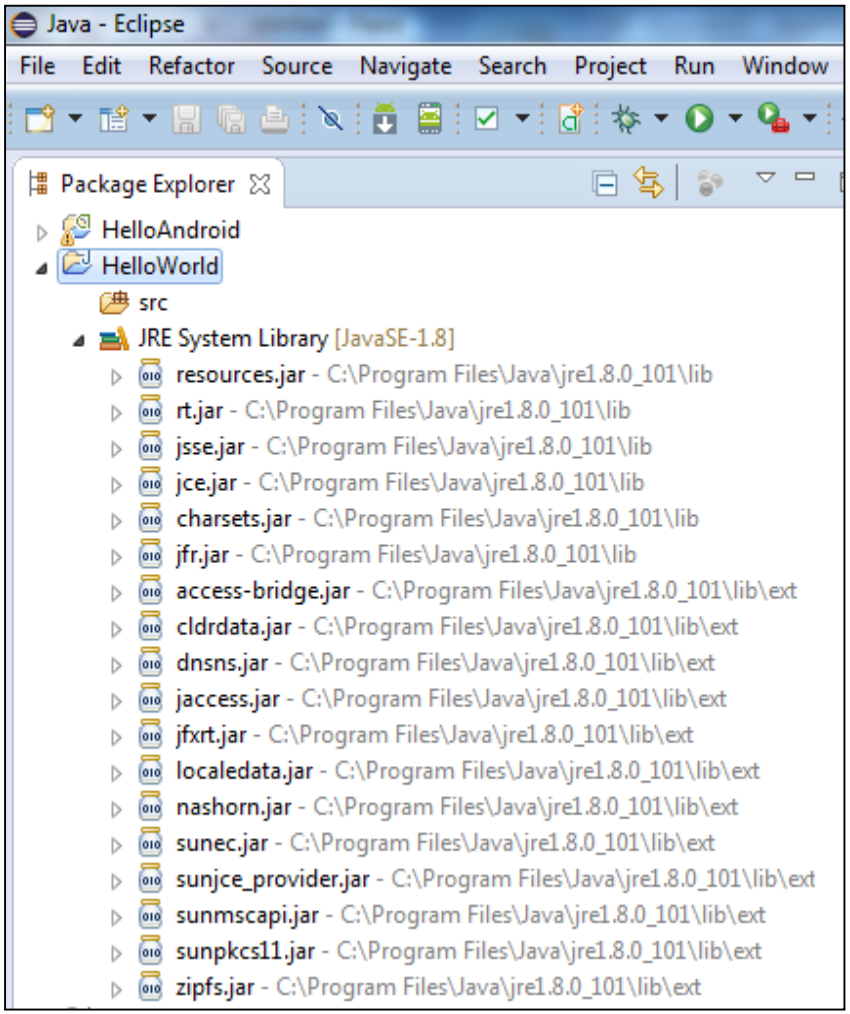
6) When handing in your assignments, you must include all of the information related to your project. For most projects, this means that both the `src`, the `.classpath`, and *all* of the files, folders and packages involved in a program: these must all be included. This can, and should, be done in the form of a single .zip file exported from Eclipse according to the instructions that will be provided to you in your assignments.

Note however that requirements may vary depending on the assignment, and only the `src` folder, zipped from inside Eclipse, may be required. You will be given precise instructions on how your assignments should be submitted; please be sure to follow these instructions exactly. In particular, remember that a *project* and a *package* are two different things.

2.3 JRE System Library

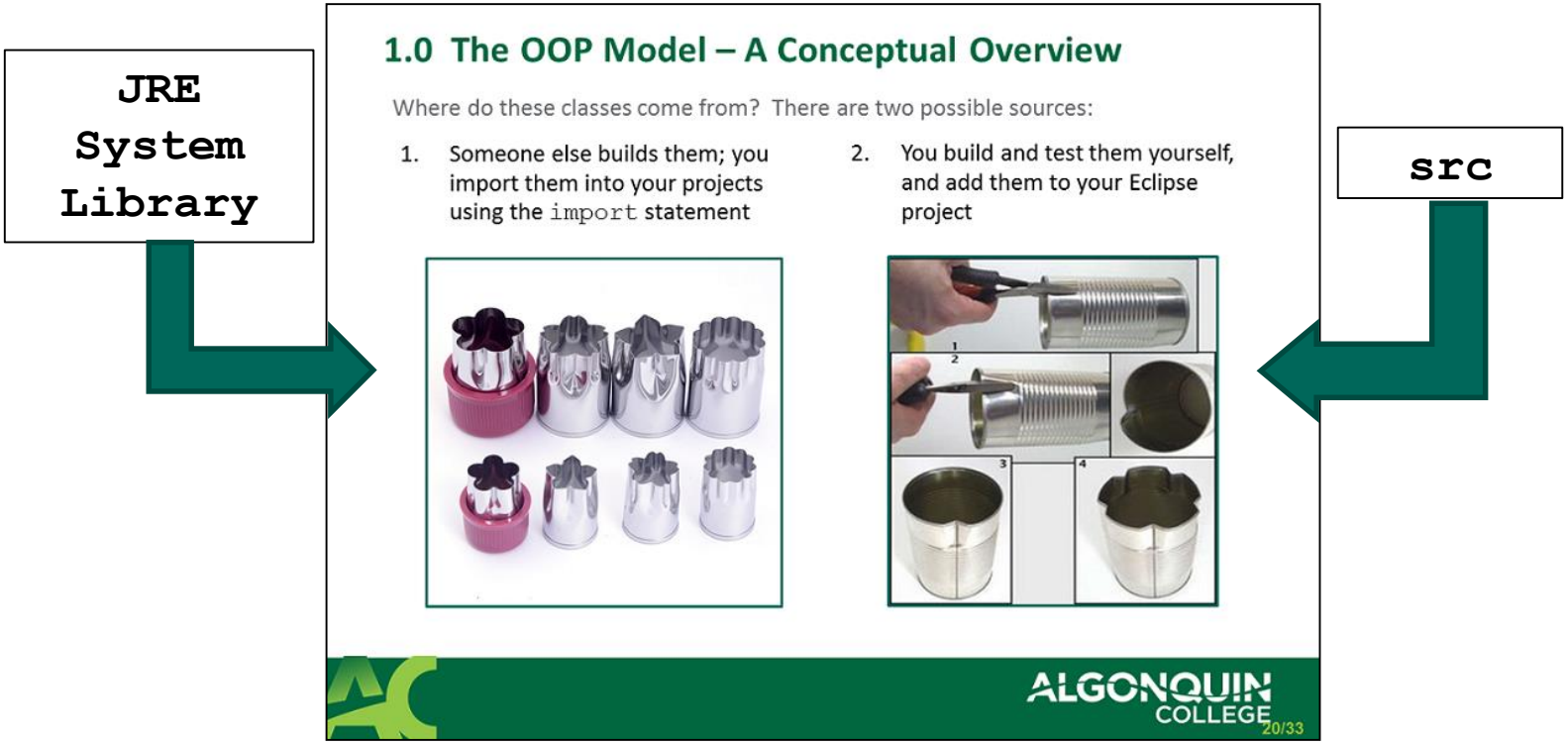
Packages are widely used to store the many classes found in Java's libraries. Consider the two folders found in the HelloWorld Project folder.

1) the `src` folder, shown in the previous slides. `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 in the `.classpath` file, which is one of the two files loaded with the project itself, as seen earlier.



2.3 JRE System Library

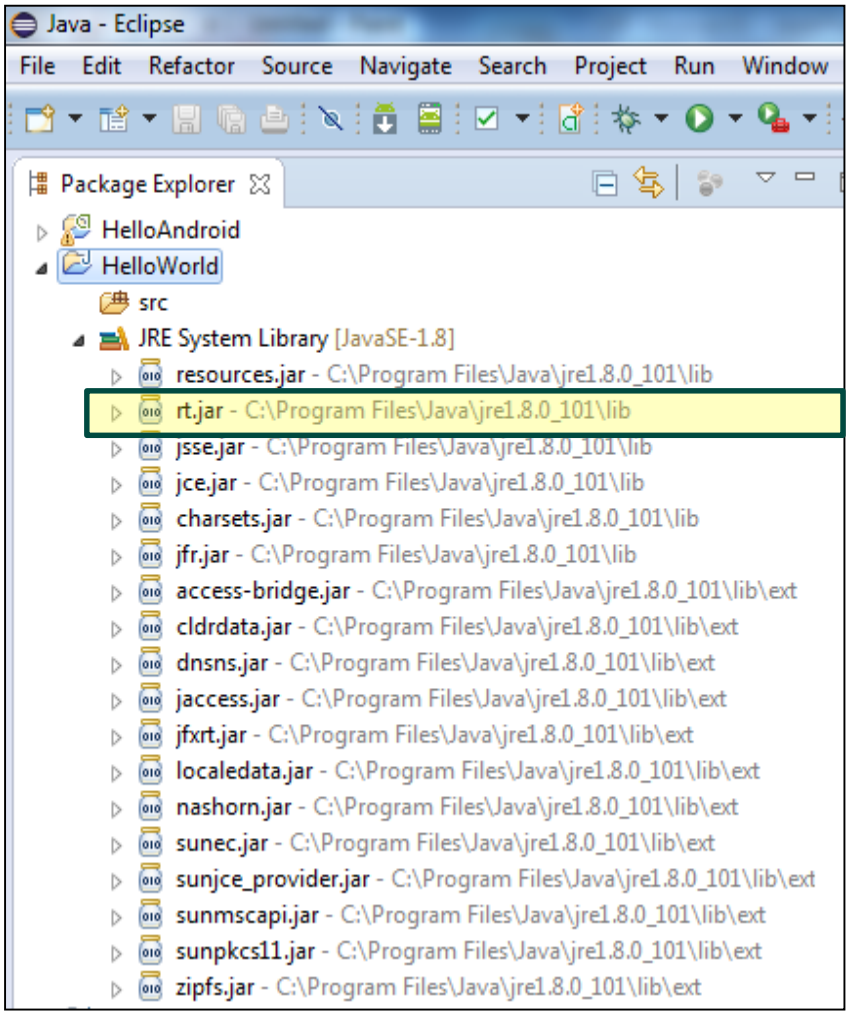
As an aside, note these two folders reflect something mentioned in Module 01, that each program is a combination of built-in library code, plus your own software. As its name suggests, the `JRE System Library` contains Java's many libraries, while `src` is where you classes will be stored.



2.3 JRE System Library

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

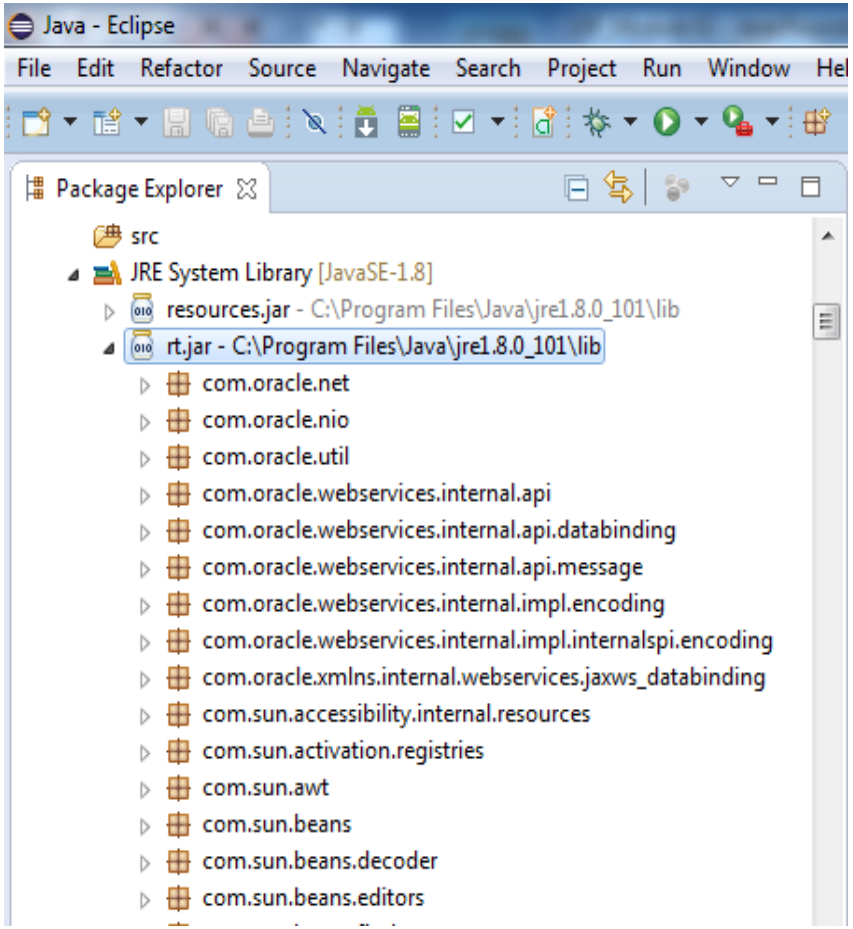
We are particularly interested in the rt.jar file, shown as the second item in the list at right. This contains, amongst other things, the default package of classes that allow your Java project to execute 'Hello World'.



2.3 JRE System Library

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

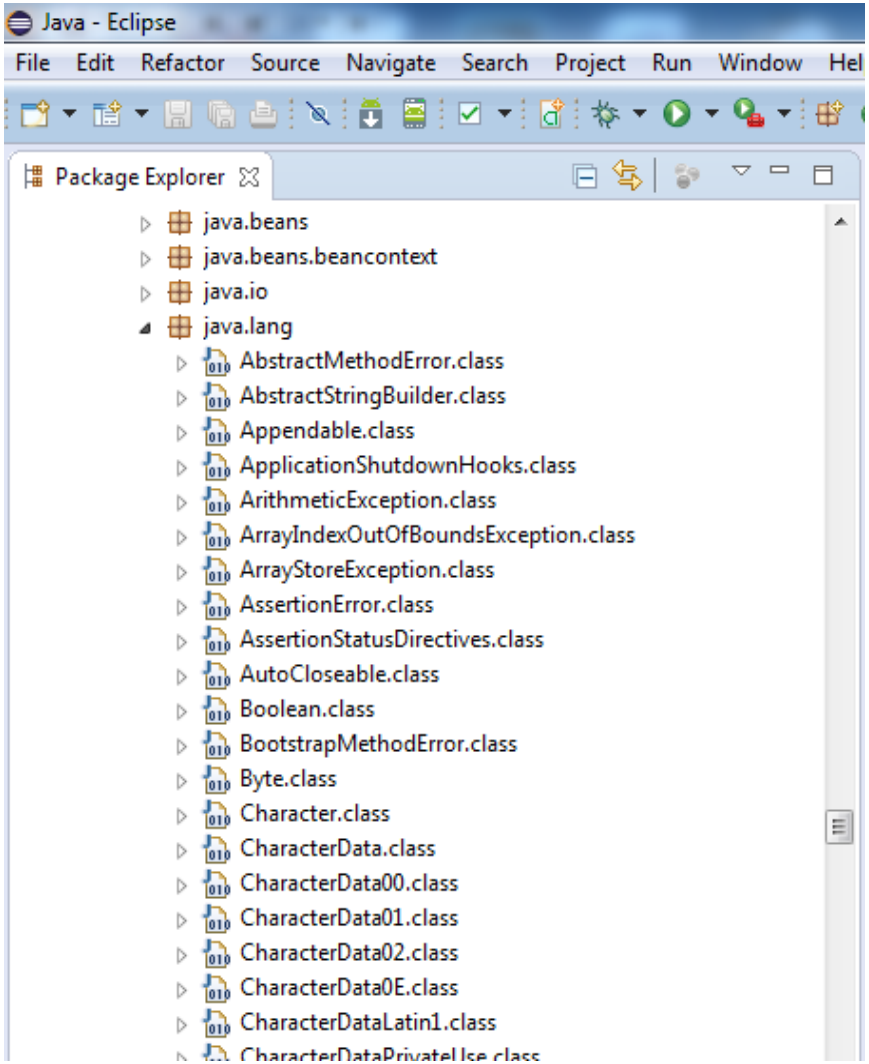
The icon for a package is a yellow box with a cross-hatch on it.



2.3 JRE System Library

Opening up a package icon, such as `java.lang`, shows 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 in your program. Rather, Eclipse is showing a list of code it has access to, which can then be explicitly loaded by your programs, as required.

The reason for this strategy should be obvious: 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 anyway.



2.3 JRE System Library

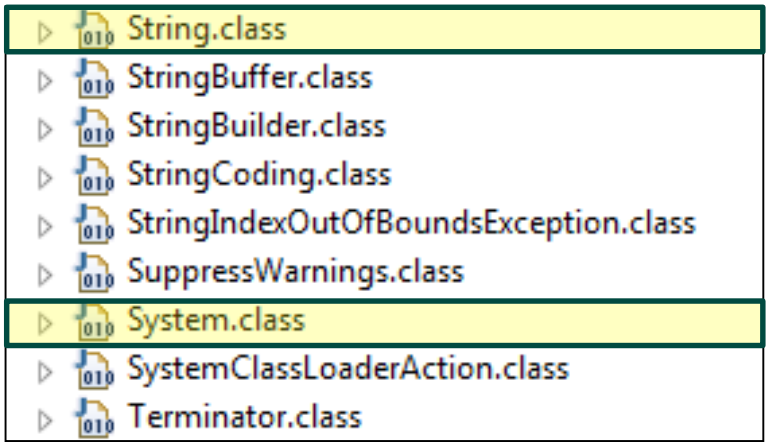
Two packages are automatically loaded by Eclipse whenever a new project is created:

a. `java.lang`

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

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

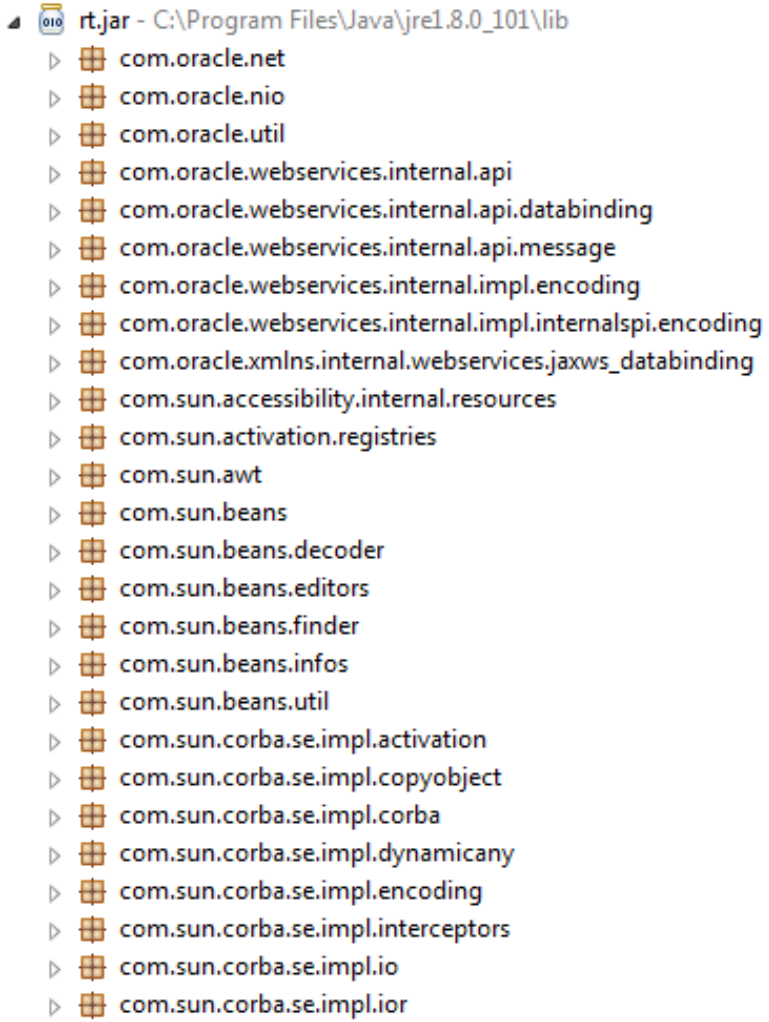
java.lang (loaded by default):



2.3 JRE System Library – Notes

1) 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 originated with the sun.com web site.

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



2.3 JRE System Library – Notes

2) 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 `lang` 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 (which, remember, are just subfolders inside `src`) 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.

2.4 Classes

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

All java code is contained in classes. 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:

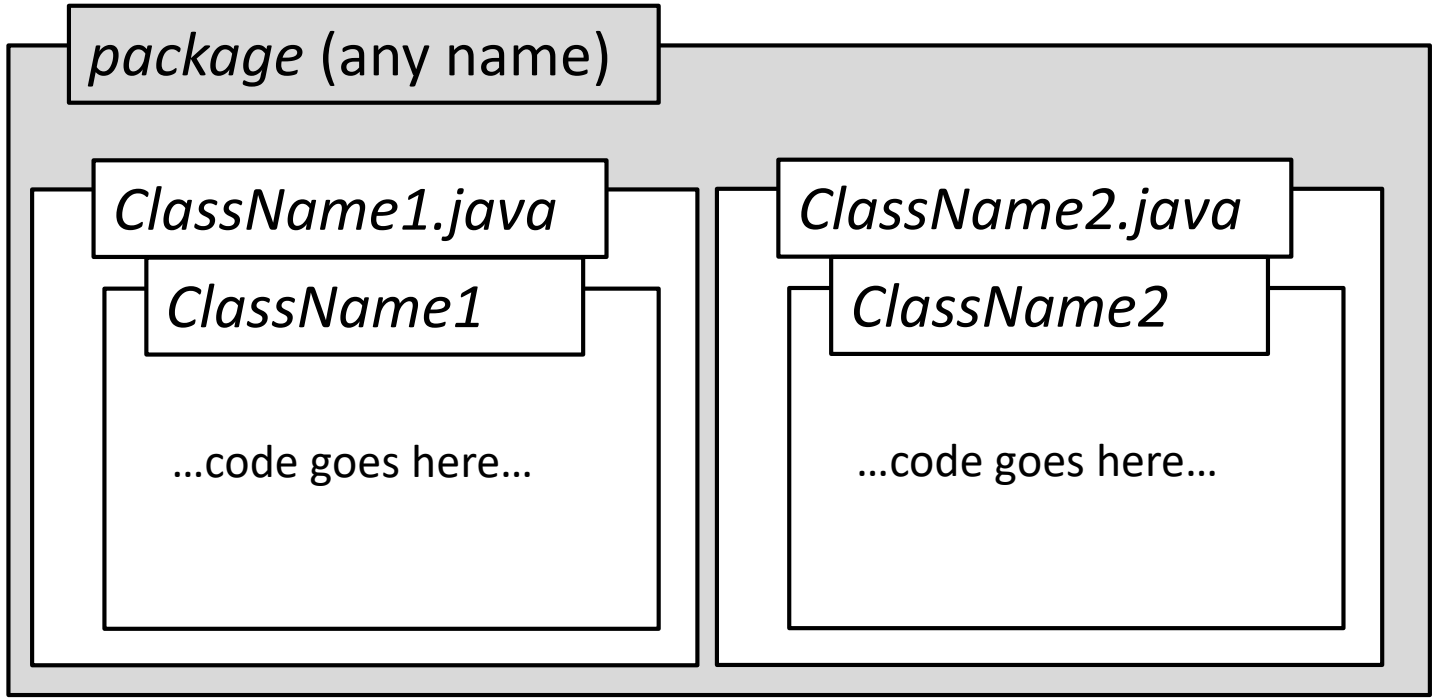
```
public class ClassName {

    …code goes inside here…

}
```

The class name must be the same as the java file in which it is stored. By convention, class names are capitalized, along with each succeeding word in the identifier.

Liang 9 | D&D 3

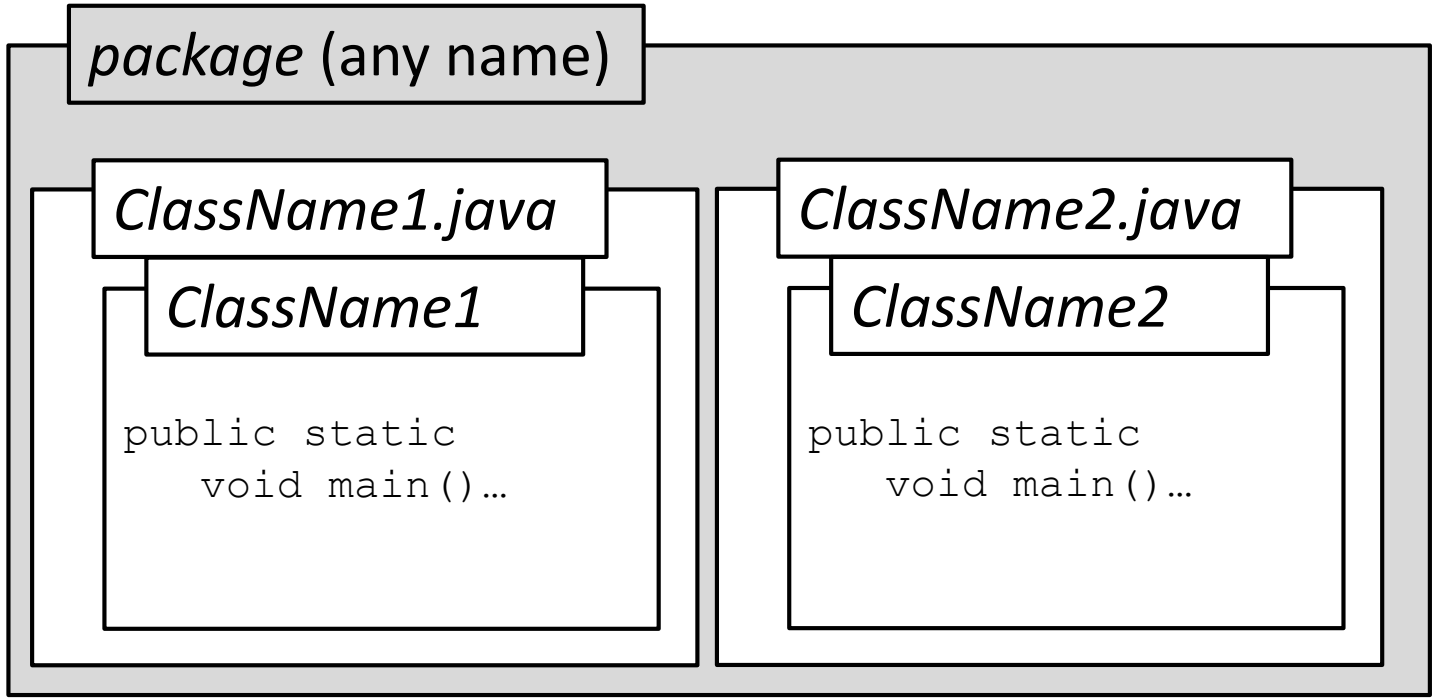
2.4 Classes

Note that each project may contain more than one package, and each package may contain more than one Java file. Each class must be the same as the name of the Java file in which it resides.



2.4 Classes

Any class with a `main()` method can act as the starting point for execution. You can call a *specific* `main()` method by citing its particular `ClassName` at the command line. In Eclipse, setting the insertion point into any class with a `main()` method before clicking the  icon has the same effect.



2.4 Classes

Each class contains fields and methods*, along with access modifiers that determine their visibility.

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

    [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.

2.4 Classes

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

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. Public methods are even visible in other packages.

Protected access means that the instantiated members are public within a package, private otherwise—with exceptions based on inheritance. 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**.

2.5 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, which is part of the `java.util` library. The code looks like this:

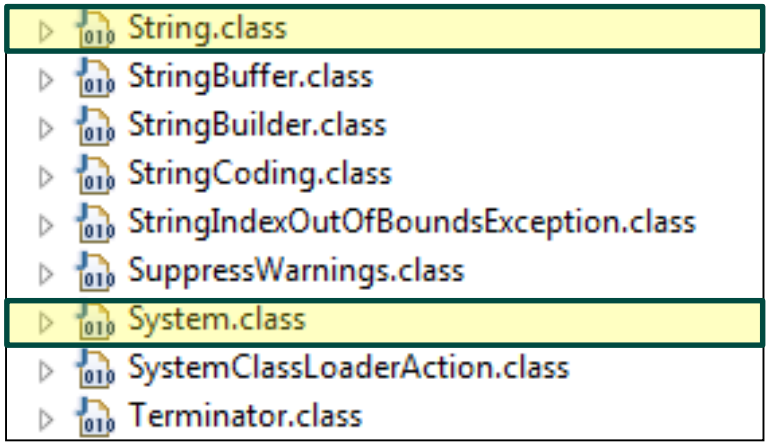
```
package cst8284.helloWorld;
import java.util.Scanner;

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

Liang 2.2 | D&D 2.5

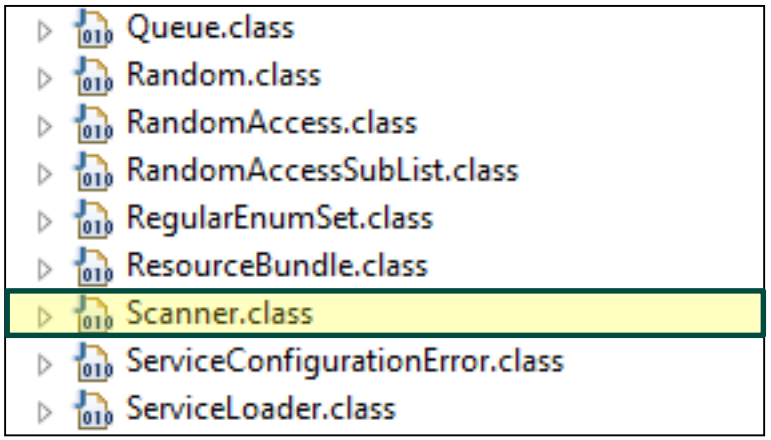
2.5 Importing Classes

Note the `import` declaration. The program imports `java.util` since, while `java.lang` is imported by default, the `java.util` package is *not*. To access the classes in this package, including `Scanner`, you must declare `Scanner` correctly to the compiler by using `import java.util.Scanner;`

java.lang (loaded by default):



java.util (must be imported):



2.5 Importing Classes

Contrary to expectations, `import` *does not* import the `Scanner` code into your Java project for inclusion in the compiled product.

In Java, each class must be identified by its **fully qualified name**, which acts like a path name that tells the compiler where to locate the bytecode associated with a class. Thus the following is valid (and often seen in code):

```
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 class like `Scanner`. So

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

tells the compiler: whenever you see `Scanner`, prefix `java.util` to the front of it.

So `import` imports *no* code into your program; it is merely a shortcut for programmers, designed to make their life easier.

2.5 Importing Classes

The wildcard symbol, *, can be used to connect any type, like `Scanner`, to the type with the matching name in a particular library. This is useful if you wish to use a number of classes from the same library in your program, but somewhat wasteful otherwise. Thus

```
import java.util.*;
```

says: for any class identifier in code that has a matching type in the library, preface the identifier with `java.util`. Since `Scanner`, `Calendar`, and `ArrayList` are all found in `java.util`, the above code is a shortcut that saves us having to type

```
import java.util.Scanner;
import java.util.Date;
import java.util.ArrayList;
...
Scanner scan = new Scanner(System.in);
Date date = new Date();
ArrayList<Date> ar = new ArrayList<Date>();
```

2.5 Importing Classes

Note that the '*' does not apply to packages in other libraries that begin with the same path name.

Thus `import java.lang.*` does not cover classes found in `java.lang.annotation`, `java.lang.instrument`, `java.lang.invoke`… or any of the other libraries that begin with `java.lang`. To access these libraries, you would need to explicitly reference `java.lang.instrument.*`, etc.

This is important in JavaFX, since most of the libraries we'll be using begin with `javafx.scene` or `com.sun.javafx`. You cannot save yourself time by simply importing `javafx.*` and `com.sun.javafx.*`; you must specify library names in their entirety in your `import` statements.

And for large JavaFX projects, this means your code will typically have dozens of `import` statements.

2.6 Constructors

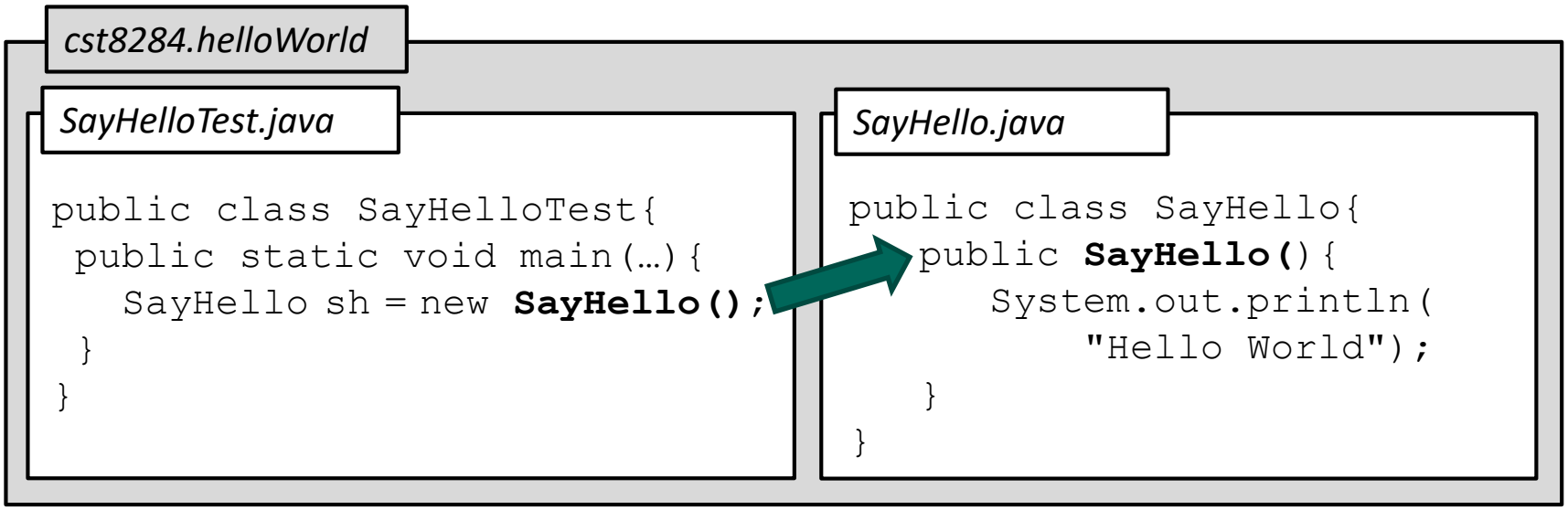
Before proceeding, we'll simplify our `SayHello` class somewhat. We now assume there are two classes, each in their own Java file:

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

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

Liang 9 | D&D 8

2.6 Constructors

In this version of "HelloWorld", the `main()` method is located in `SayHelloTest`. It contains a single line of code, which instantiates a new `SayHello` object, whose constructor prints out “Hello World” when the new `SayHello` object is instantiated.



Liang 9.4

D&D 8.5

2.6 Constructors

`SayHello()` is 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.

cst8284.helloWorld

***SayHello**.java*

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

2.6 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 just fine if we write:

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

2.6 Constructors

When any two class members (including constructors, which act like 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:

cst8284.helloWorld

***SayHello**.java*

```
public class SayHello{

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

Liang 6.8 | D&D 6.12

2.6 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

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

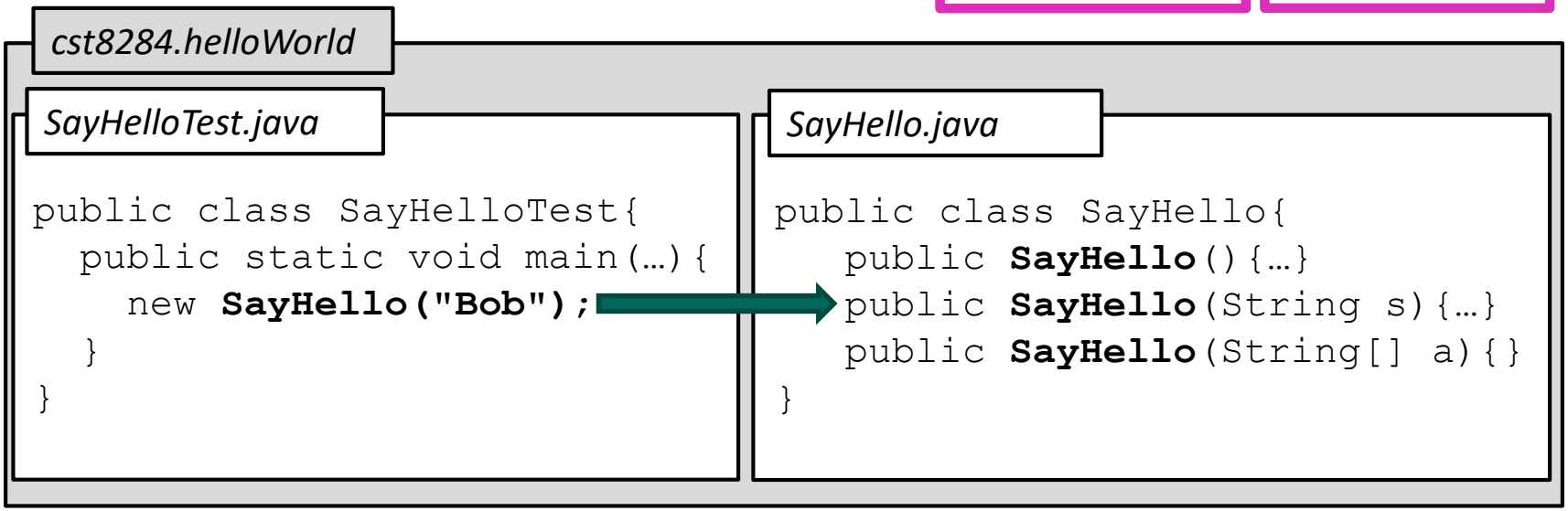
2.6 Constructors

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



2.6 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.

cst8284.helloWorld

SayHello.java

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

2.6 Constructors

Rather than reproduce *almost* the same code twice, we can call the second constructor from 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'. We do this using **`this`**, where *`this` represents a current class name*. In the example below, `this()` *means* `SayHello()`. But note: in Java, you cannot refer to the classname directly inside the class: you must use `this`.

cst8284.helloWorld

***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

2.6 Constructors

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

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

which clearly indicates that the `SayHello(String s)` constructor is invoked.

cst8284.helloWorld

SayHello.java

```
public class SayHello{

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

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

2.6 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 use this constructor to output multiple "Hello" statements, and then chain this constructor to the other two. The `SayHello()` code needed to display the contents of the array is:

cst8284.helloWorld

SayHello.java

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

2.6 Constructors

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

****Note! An array of `Strings` is an *`array`* object; it is *not* a `String` object.
Thus the last two constructors have different signatures*****

cst8284.helloWorld

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]);
   }
}
```

2.6 Constructors

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

cst8284.helloWorld

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]);
   }
}
```

2.6 Constructors

In the four instantiations of a `SayHello` object in the `SayHelloTest` code shown below, the constructor actually invoked will be the one whose signature best matches the three `SayHello` constructors on the previous slide. However, regardless of which constructor is initially called, it is the third constructor that eventually does all the actual work.

cst8284.helloWorld

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
   }
}
```

2.6 Constructors

The complete, final version of this iteration of 'Hello World' is shown below, on this and the next slide.

In `SayHelloTest.java` we will have:

```
package cst8284.helloWorld

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

2.6 Constructors

In the SayHello.java file we will have:

```
package cst8284.helloWorld
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[] strAr){
      for (int i=0; i<strAr.length; i++)
         System.out.println("Hello " + strAr[i]);
   }
}
```

2.6 Constructors

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, and it takes longer to execute than just building each constructor separately. But…

2.6 Constructors

…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.)

2.6 Constructors

In the toy example that follows, 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, we assume the employee's full name and/or employee number has already been obtained and the 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 of a spreadsheet.

The last constructor, the one the previous constructors are linked to, uses an overloaded, chained method called `setFullName()` to handle four different variations on the way a person's name might be entered, i.e. as last-name only, first-middle-last, first-middle initial-last, or first-last.

2.6 Constructors

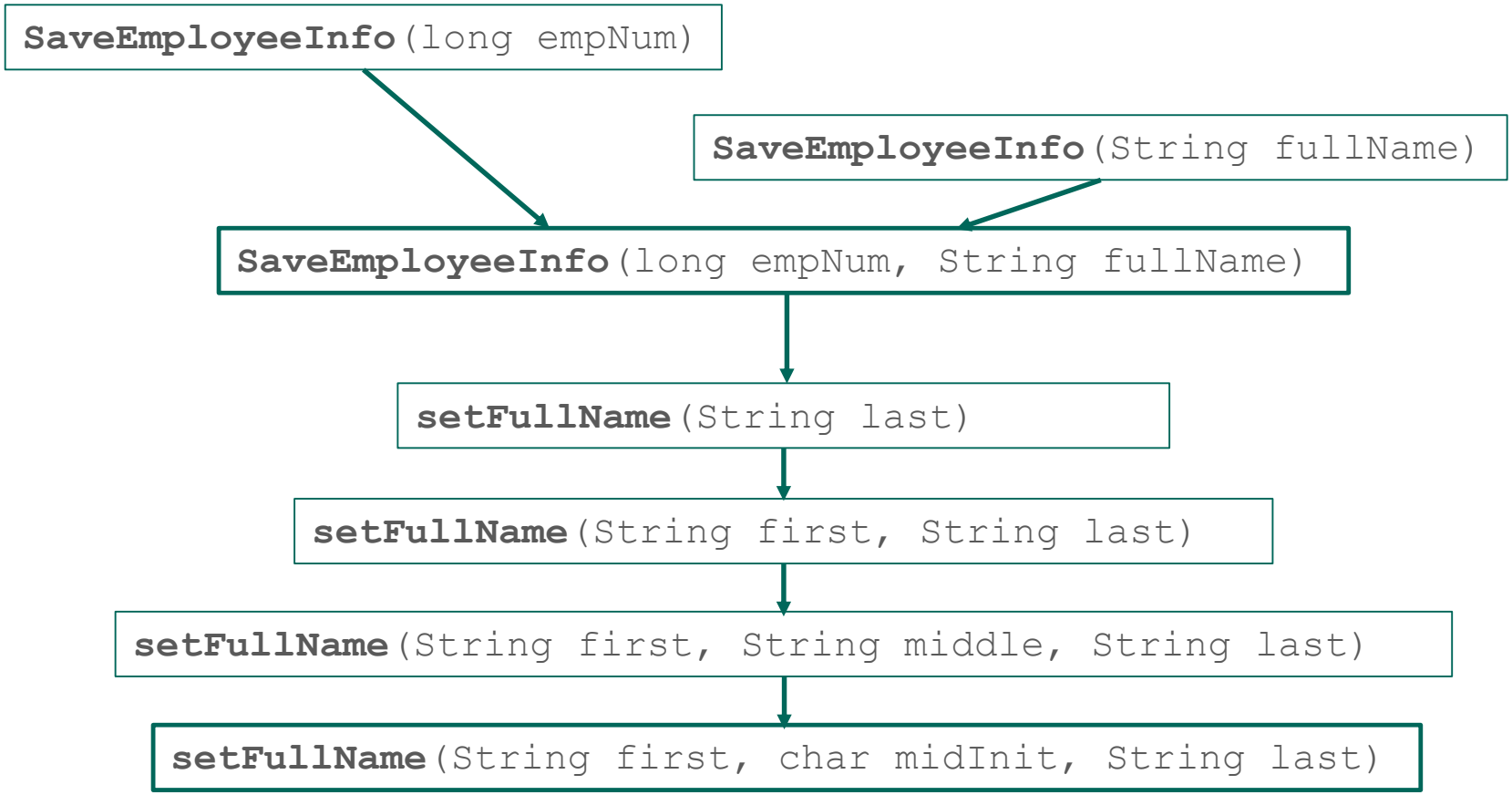
```
public class SaveEmployeeInfo{

   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 info"); exit(0);
          case 1:  setFullName(sAr[0]); break;
          case 2:  setFullName(sAr[0], sAr[1]); break;
          default:
                   setFullName(sAr[0], sAr[1], sAr[LAST]); break;
      }
   } …
```

2.6 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);
  }
}
```

2.6 Constructors

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



2.6 Constructors

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`.

2.6 Constructors

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.

(Note that ambiguously overloaded methods like the one at the top of the screen are to be avoided, since nothing clearly distinguishes the actual function of the two methods. Their lack of a distinct functionality makes them dangerous: how would the user possibly guess which one to use, given that their signatures are almost identical, except for the order?)

2.6 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 new method that has been added to the class.
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 that object (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. Additionally, for overloaded methods, the signature does not depend on the return type. This somewhat trivial fact is useful to know for Module 04, when we discuss subclasses and overriding.

2.6 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—like the no-arg `SaveEmployeeInfo()` constructor in the above example.

2.6 Constructors – Notes

6. Constructors do *not* have return types. Its

```
public HelloWorld()
```

and *not* (a common mistake)

```
public void HelloWorld()
```

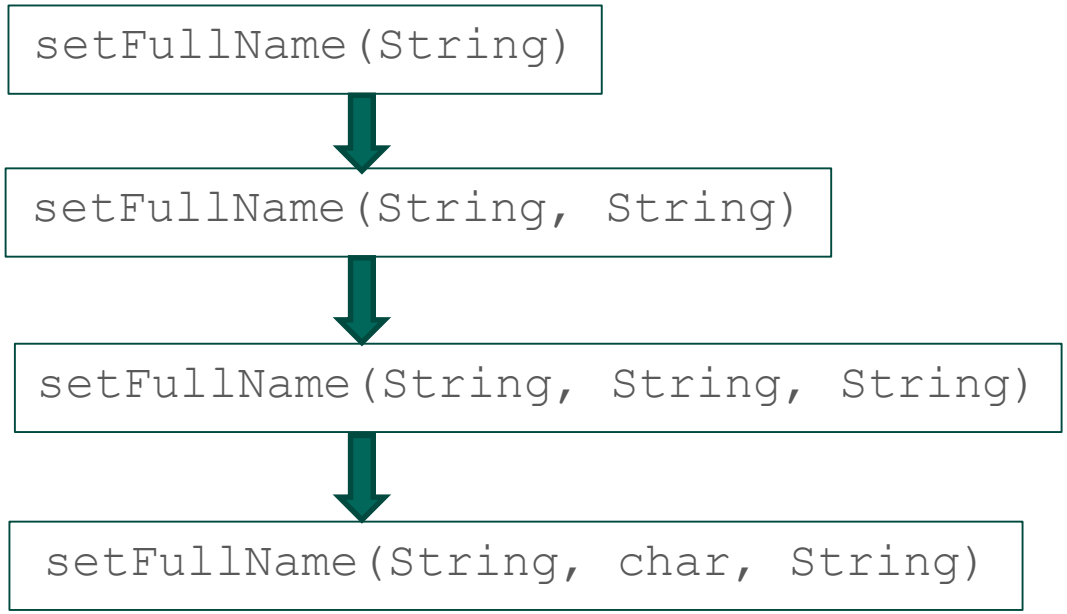
Therefore, you cannot treat a constructor method the same as an instantiated method; you cannot write

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

Constructors in Java are special, and are only executed when the object is instantiated with `new`; they cannot be 're-executed' afterward in the way a normal method can be called repeatedly.

2.6 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):



2.6 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 constructors would be written:

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

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

Questions

1. If you save a java class in a package called `myPack`, then, after you've run your code for the first time, where would you expect to find its .class file?
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 exactly 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?

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?