Access Control

Access control

- If you leave a piece of code in a drawer for a while and come back to it, you may see a much better way to do it
  - This is one of the prime motivations for *refactoring*
  - Rewrite working code in order to make it more ***readable***, ***understandable***, and thus ***maintainable***
- You want to change it; consumers (client programmers) want it to stay the same
- Separate the things that change from the things that stay the same

Access control (Cont.)

- Java provides ***access specifiers*** to allow the library creator to say what is available to the client programmer and what is not
  - The levels of access control from "most access" to "least access" are ***public***, ***protected***, ***package access*** (which has no keyword), and ***private***
- There's still the question of how the components are bundled together into a cohesive library unit
  - This is controlled with the ***package*** keyword in Java
  - The ***access specifiers*** are affected by whether a class is in the same package or in a separate package
- Learn how library components are placed into packages and understand the complete meaning of the access specifiers

package: the library unit

- A package contains a group of classes, organized together under a single namespace
  - utility library - java.util (ArrayList)

```
1 public class FullQualification {
2   public static void main(String[] args) {
3     java.util.ArrayList list = new java.util.ArrayList();
4   }
5 } ///:~
```

  - Use the *import* keyword instead

```
1 import java.util.ArrayList;
2
3 public class SingleImport {
4   public static void main(String[] args) {
5     ArrayList list = new java.util.ArrayList();
6   }
7 } ///:~
```

  - To import everything, you simply use the '*'
    - E.g., *import* java.util.*;

package: the library unit (Cont.)

- **The names of all your class members are insulated from each other**
  - **E.g., Method name:**

    A method f( ) inside a class A will not clash with an f( ) that has the same signature in class B

- ***import*: a mechanism to manage class namespaces**

- **Potential clashing of Class names**
  - ***Solution*: create a unique identifier combination for each class**
  - **Example**
    - **import edu.seu.ch01.test.*;**
    - **import edu.seu.ch02.test.*;**

Code organization

- Compile a *.java* file and may get a few *.class* files
- A Java working program is a bunch of *.class* files, which can be packaged and compressed into a Java ARchive (*JAR*) file (using Java's jar archiver)
- Java interpreter is responsible for finding, loading, and interpreting these files
- A library is a group of these class files
  - Each source file usually has a *public* class and any number of *non-public* classes
  - All these components belong together *package*

Code organization (Cont.)

- **Use a *package* statement**
  - ***Must*** **appear as the first non-comment in the file**

```
package access;
```

  - **The *public* class name within this compilation unit is under the umbrella of the name *access***
  - **Anyone who wants to use that name must either fully specify the name or use the *import* keyword**

- **Note**
  - **Java package names is to use *all lowercase letters*, even for intermediate words**

Code organization (Cont.)

- **Example**

```
1  package access.mypackage;
2
3  public class MyClass {
4    // ...
5  } ///:~
```

- **Usage**

```
1  public class QualifiedMyClass {
2    public static void main(String[] args) {
3      access.mypackage.MyClass m =
4        new access.mypackage.MyClass();
5    }
6  } ///:~
```

Or

```
1  import access.mypackage.*;
2
3  public class ImportedMyClass {
4    public static void main(String[] args) {
5      MyClass m = new MyClass();
6    }
7  } ///:~
```

Creating unique *package* names

- ***package*** **name**
  - **The reversed Internet domain name of the creator of the class, e.g., cn.edu.seu.myDataClass**
  - **Resolving the package name into a directory on your machine**
- **Java interpreter**
  - **Find the environment variable *CLASSPATH***
  - **Search for *.class* files**
  - **Take the *package* name and replace each dot with a slash**
    - ***package*** **cn.edu.seu becomes cn\edu\seu or cn/edu/seu or possibly something else, depending on your operating system**

Creating unique *package* names (Cont.)

- **Example**

```
1  package net.mindview.simple;
2
3  public class List {
4    public List() {
5      System.out.println("net.mindview.simple.List");
6    }
7  } ///:~
```

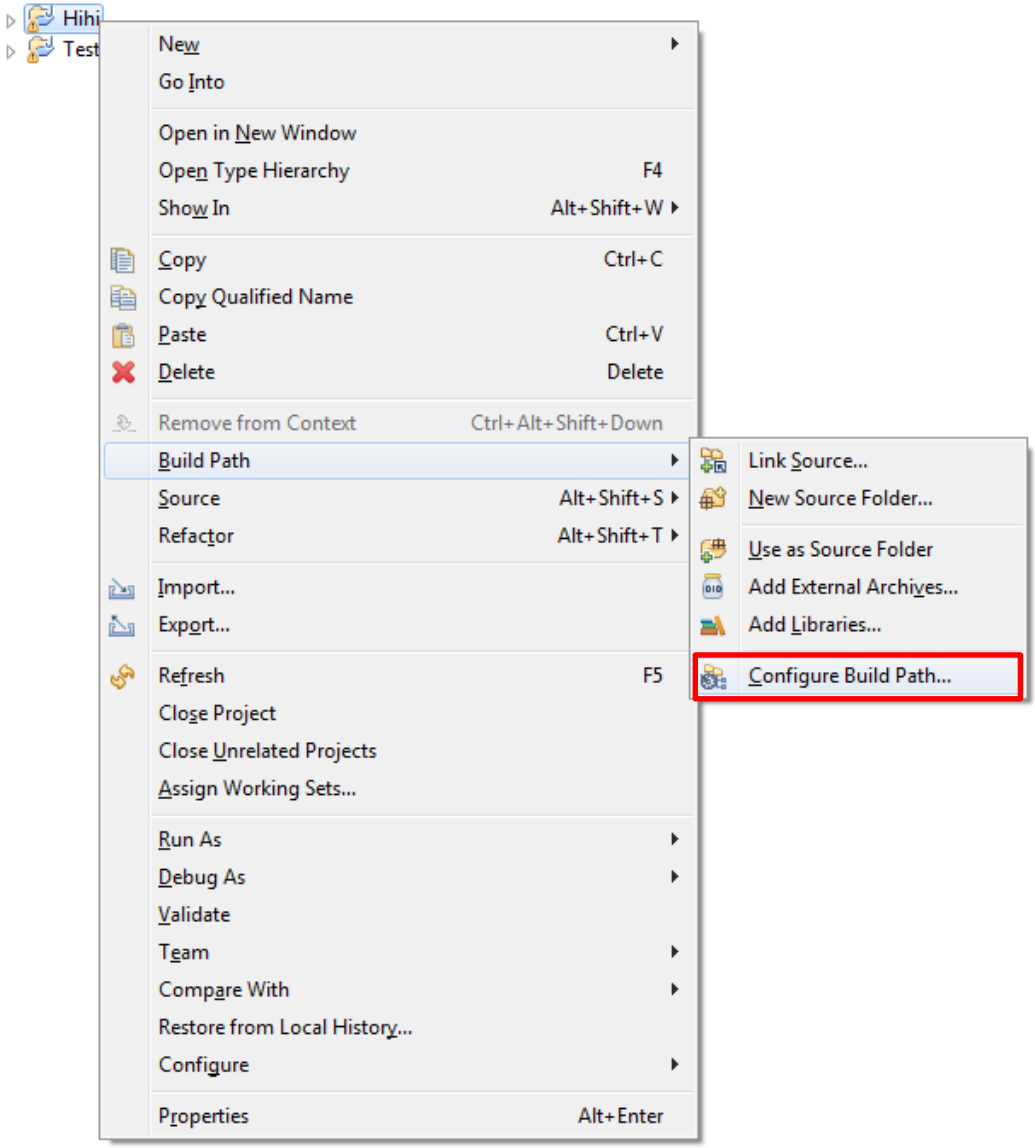
```
1  package net.mindview.simple;
2
3  public class Vector {
4    public Vector() {
5      System.out.println("net.mindview.simple.Vector");
6    }
7  } ///:~
```

- **These two files are located in E:\mylib\net\mindview\simple**
- **CLASSPATH=.;E:\mylib;C:\Program Files\Java\jre1.8.0_60\lib\rt.jar**

```
1  import net.mindview.simple.*;
2
3  public class LibTest {
4    public static void main(String[] args) {
5      Vector v = new Vector();
6      List l = new List();
7    }
8  }
```

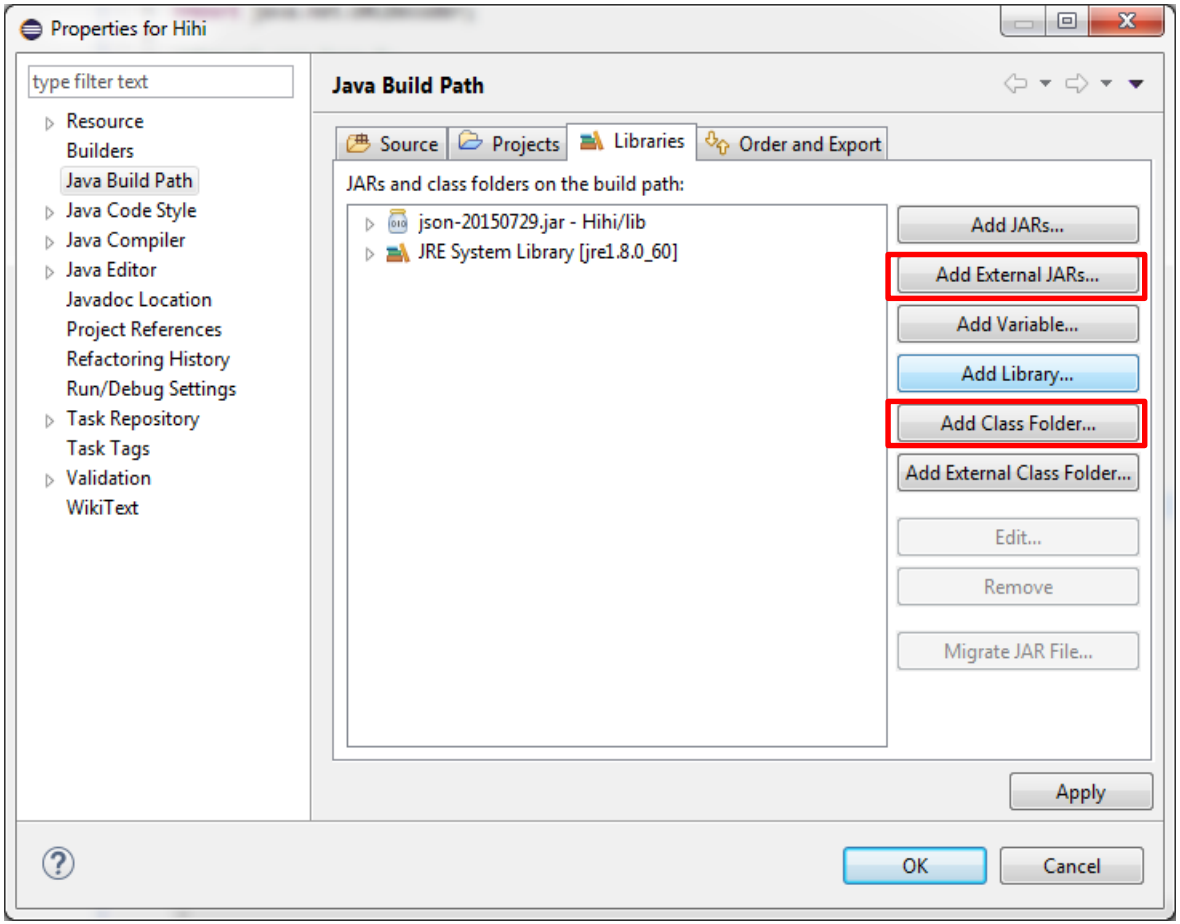
Eclipse: set classpath

- Right click your project to select the “Build Path” menu item
- Select “Configure Build Path…”



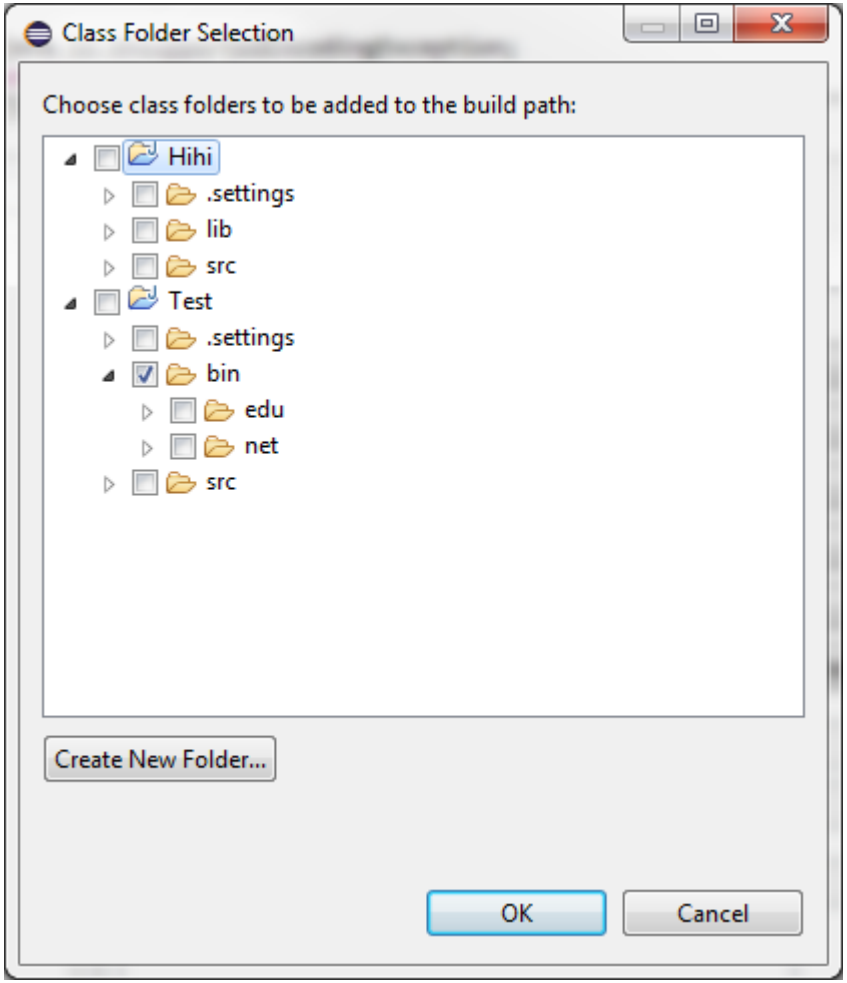
Eclipse: set classpath (Cont.)

- Select “Libraies” Tab to set the class path
- You can add External Jars or add Class Folder at there

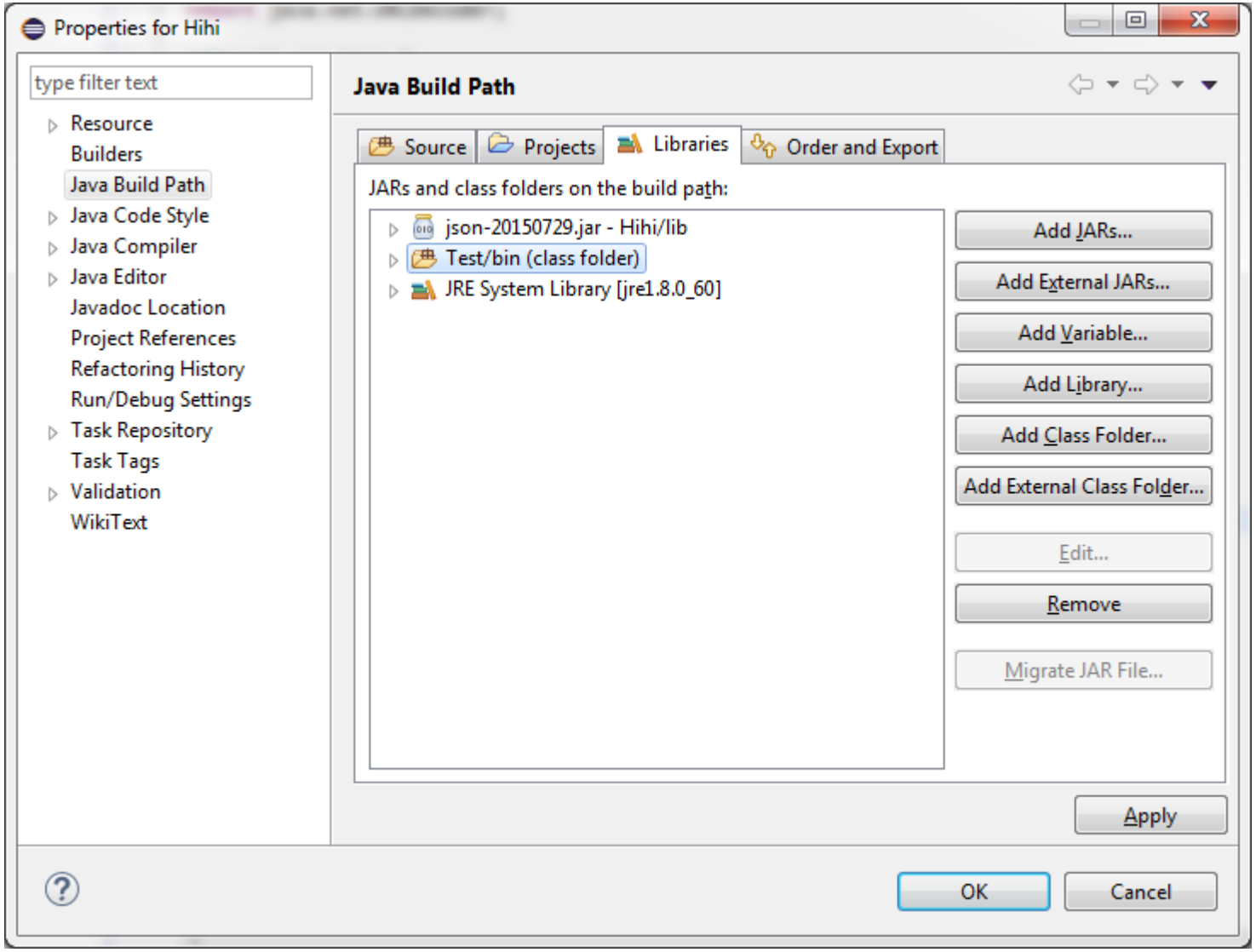


Eclipse: set classpath (Cont.)

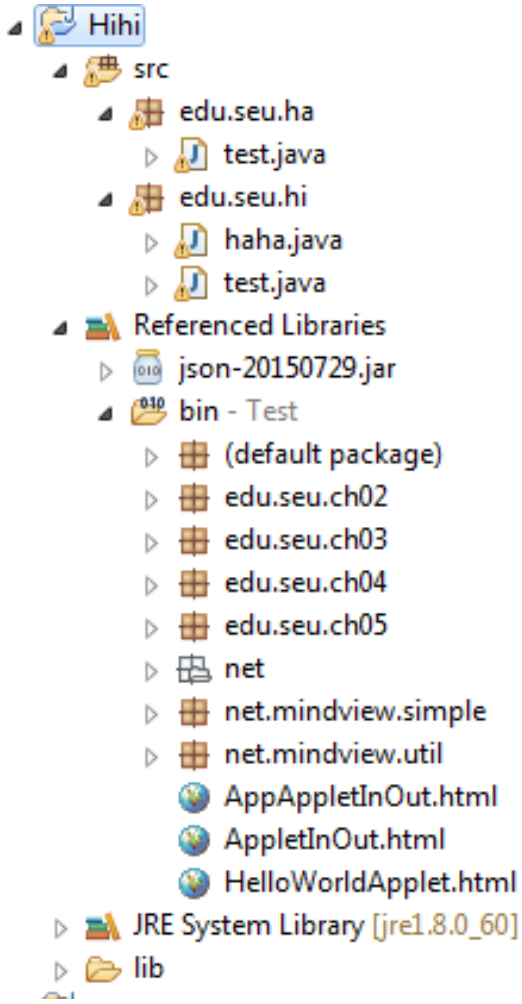
❑ Add Class Folder

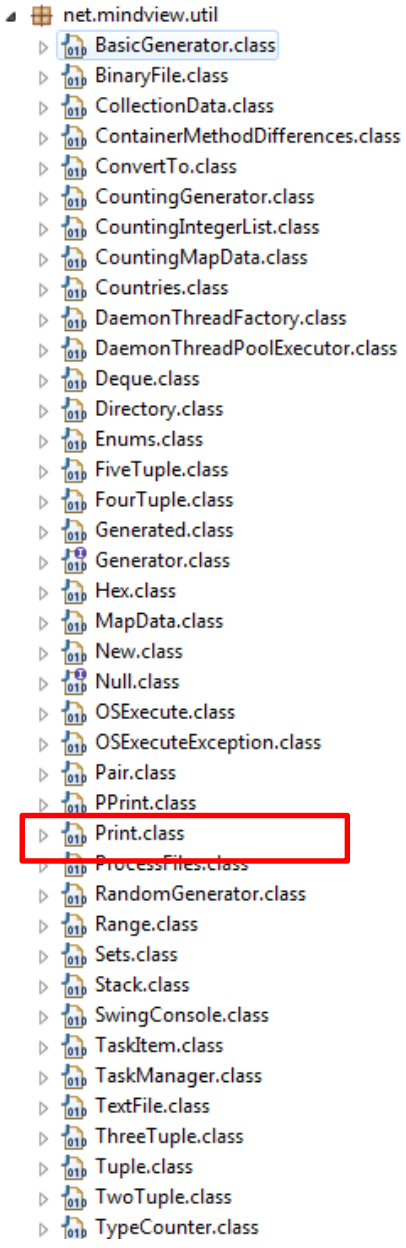


Eclipse: set classpath (Cont.)

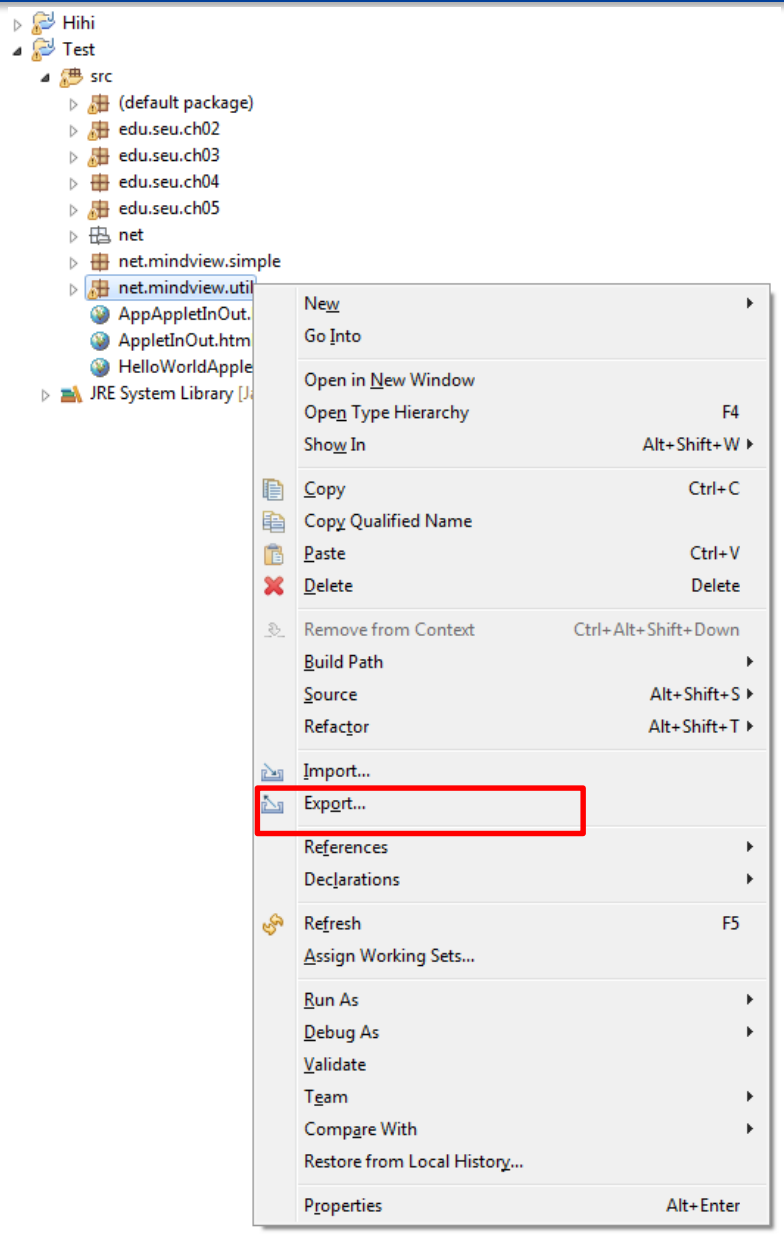


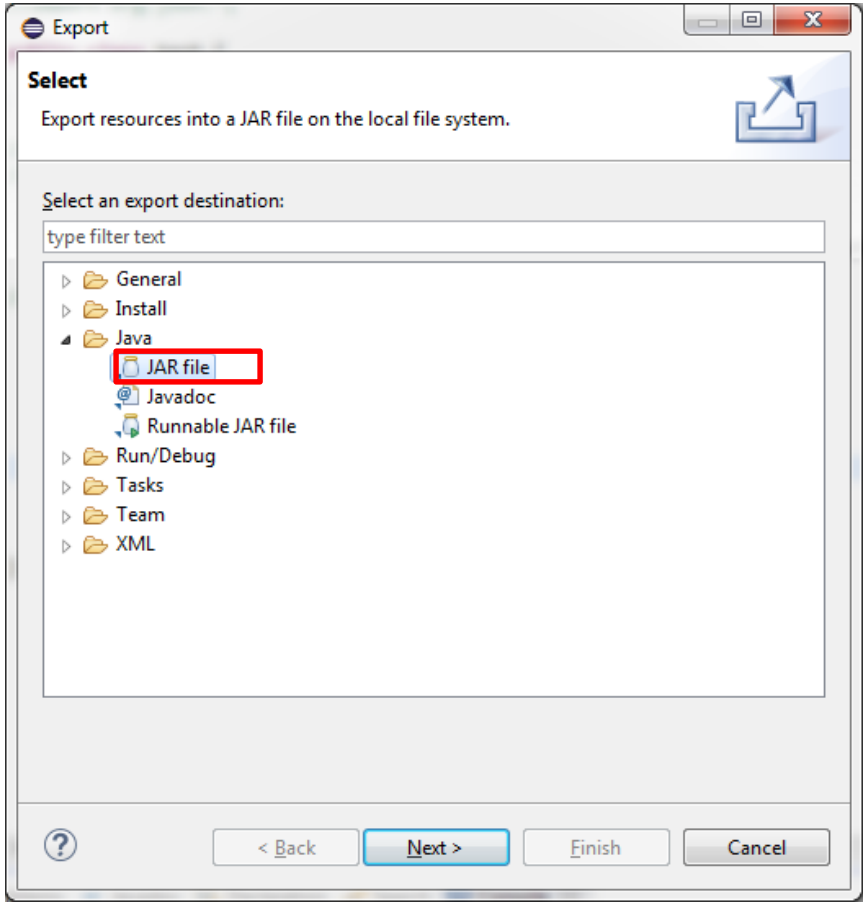
Eclipse: set classpath (Cont.)



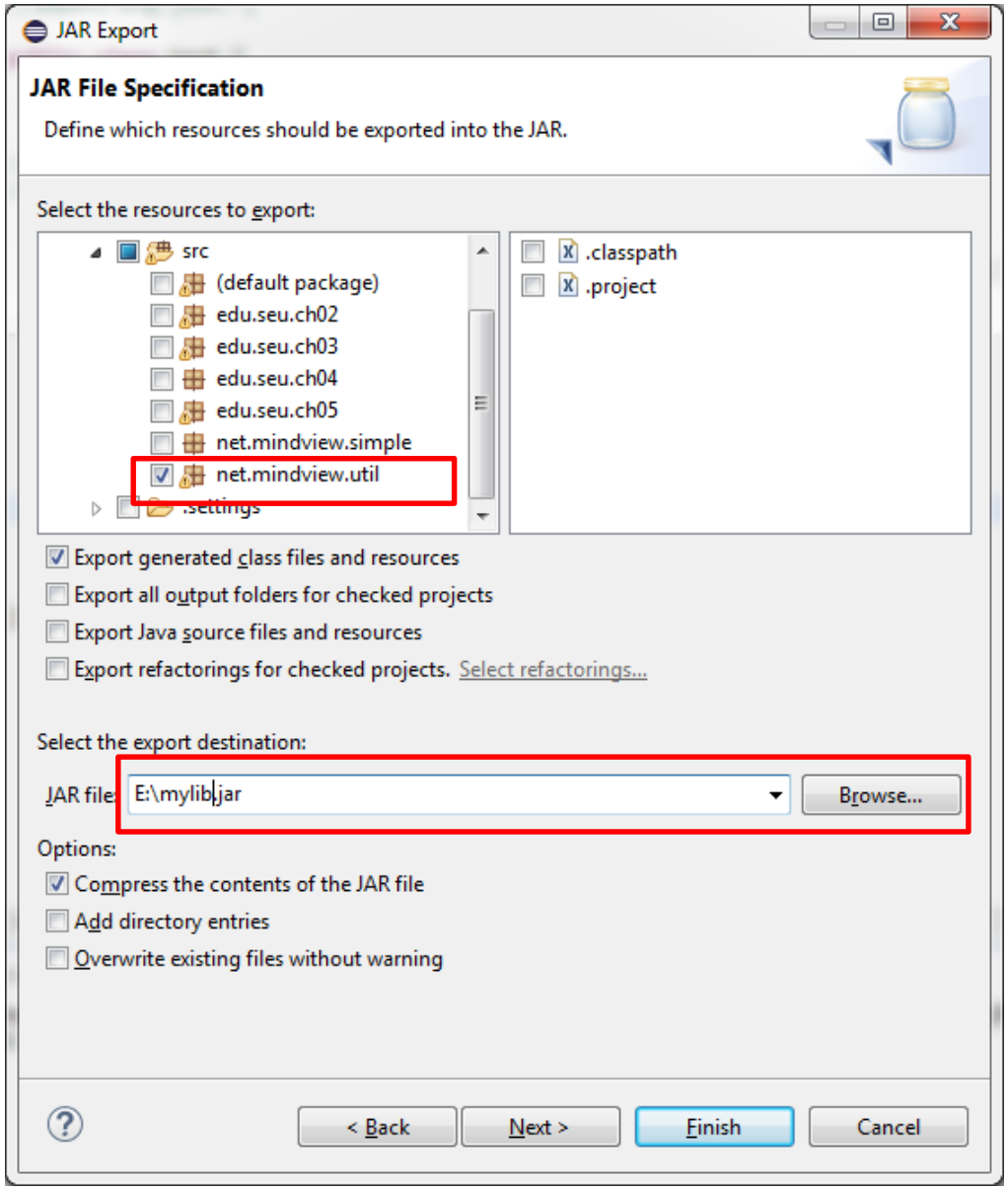


Eclipse: generate a JAR

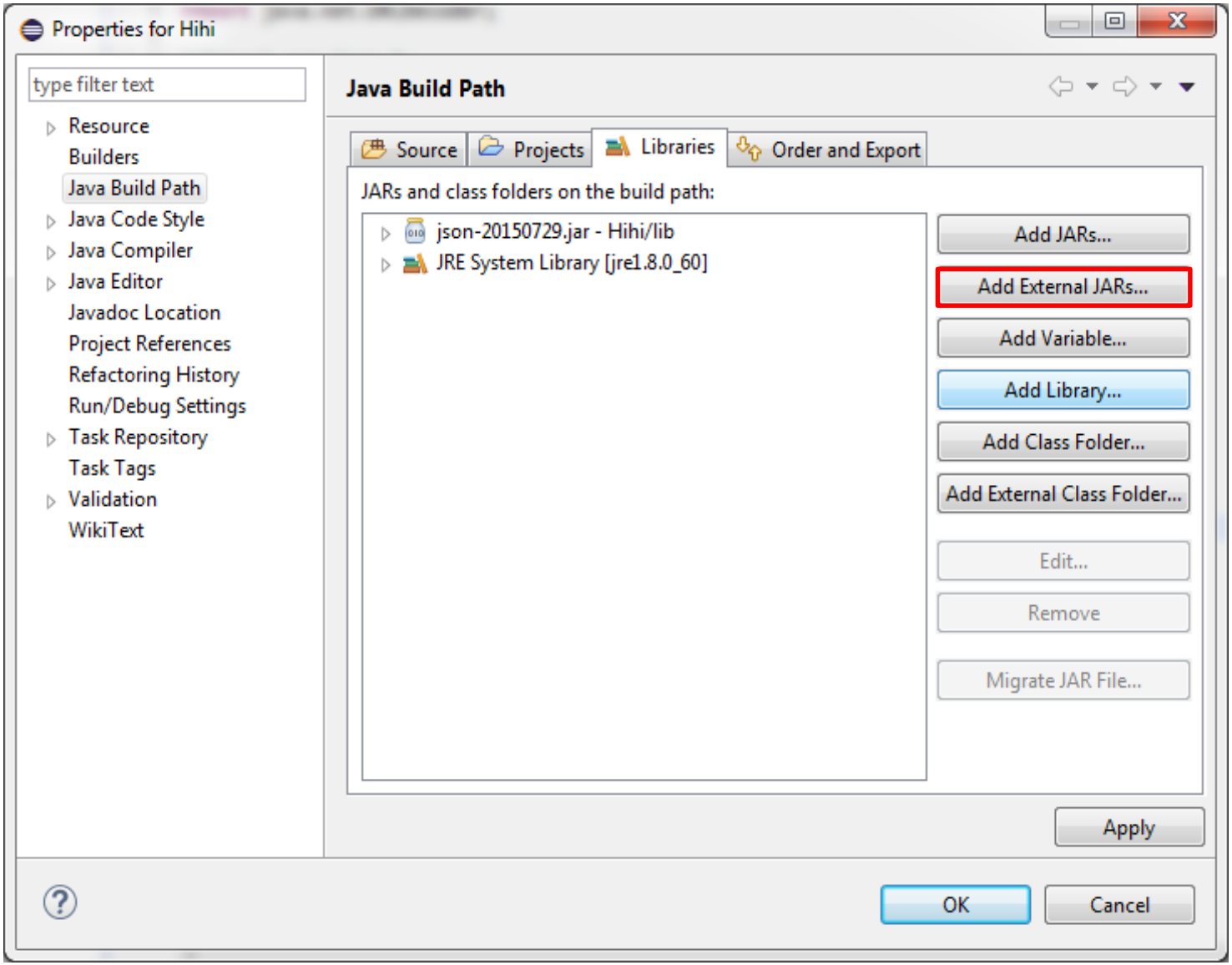




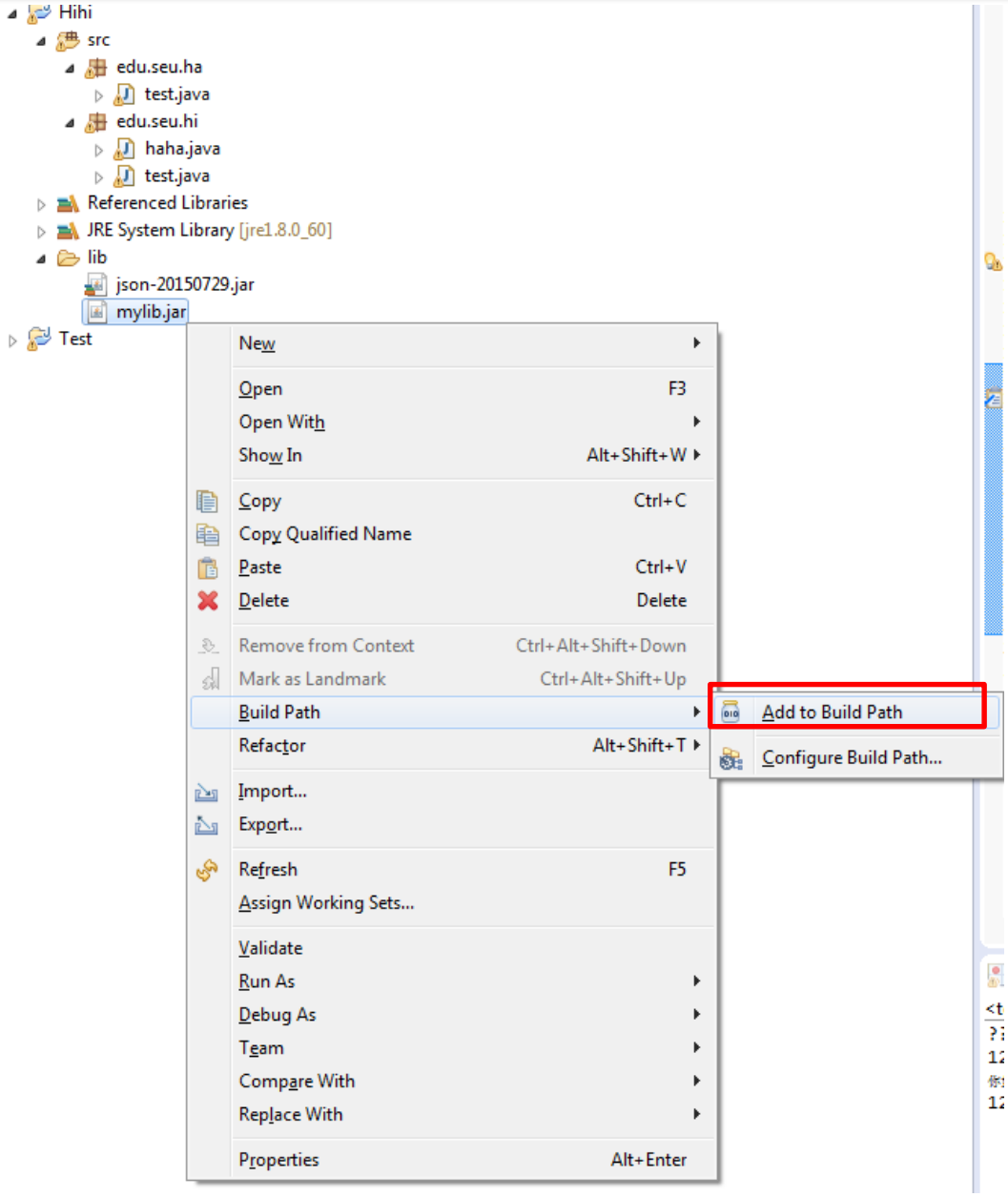
Eclipse: generate a JAR (Cont.)

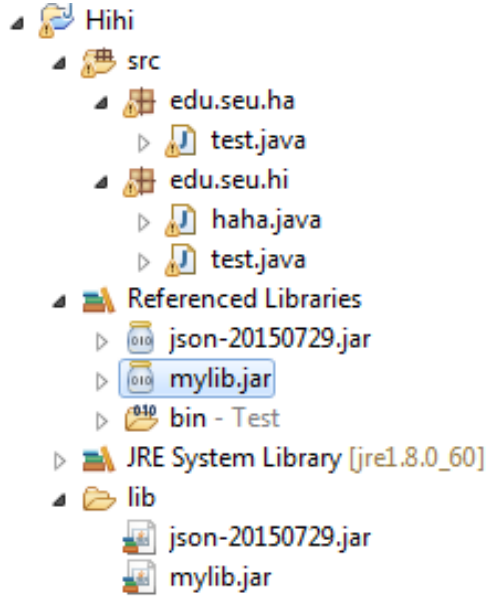


Eclipse: Add a JAR for a project



Eclipse: Add a JAR for a project (Cont.)





Collisions

- What happens if two libraries are imported via '*' and they include the same names?

```
import net.mindview.simple.*;
import java.util.*;
```

- Note

```
1  package net.mindview.simple;
2
3  public class Vector {
4    public Vector() {
5      System.out.println("net.mindview.simple.Vector");
6    }
7  } ///:~
```

- *java.util.** also contains a *Vector* class, this causes a potential collision

```
java.util.Vector v = new java.util.Vector();
```

- Alternatively, use the single-class import form to prevent clashes

A custom tool library

- **Create your own libraries of tools to reduce or eliminate duplicate code**

```
1  package net.mindview.util;
2  import java.io.*;
3
4  public class Print {
5    // Print with a newline:
6    public static void print(Object obj) {
7      System.out.println(obj);
8    }
9    // Print a newline by itself:
10   public static void print() {
11     System.out.println();
12   }
13   // Print with no line break:
14   public static void printnb(Object obj) {
15     System.out.print(obj);
16   }
17   // The new Java SE5 printf() (from C):
18   public static PrintStream
19   printf(String format, Object... args) {
20     return System.out.printf(format, args);
21   }
22 } ///:~
```

Java access specifiers

- **Java access specifiers *public*, *protected*, and *private***
  - **Placed *in front of* each definition for each member in your class**
    - **A field or a method**

- **If you don't provide an access specifier, it means "*package access*"**

Package access

- The default access has no keyword, referred to as ***package access***
- All the other classes in the current package have access to that member
- All the classes outside of this package, the member appears to be ***private***
- ***Package access*** allows you to group related classes together in a package
  - They can easily interact with each other

public: interface access

- **Make the member *public,* then everybody, everywhere, can access it**
  - **javac access\desert\Cookie.java**

```
1  package access.dessert;
2
3  public class Cookie {
4    public Cookie() {
5     System.out.println("Cookie constructor");
6    }
7    void bite() { System.out.println("bite"); }
8  } ///:~
```

  - **javac Dinner.java**
  - **java Dinner**

```
1  import access.dessert.*;
2
3  public class Dinner {
4    public static void main(String[] args) {
5      Cookie x = new Cookie();
6      //! x.bite(); // Can't access
7    }
8  }
```

The default package

- The following code would appear that it breaks the rules:

```
1  class Cake {
2    public static void main(String[] args) {
3      Pie x = new Pie();
4      x.f();
5    }
6  }
```

- In a second file in the same directory

```
1  class Pie {
2    void f() { System.out.println("Pie.f()"); }
3  } ///:~
```

private: you can't touch that!

- **No one can access that member except the class that contains that member, inside methods of that class**
  - **Other classes in the same package cannot access *private* members**
  - ***private* allows you to freely change that member without concern that it will affect another class in the same package**

```
1  class Sundae {
2    private Sundae() {}
3    static Sundae makeASundae() {
4      return new Sundae();
5    }
6  }
7
8  public class IceCream {
9    public static void main(String[] args) {
10     //! Sundae x = new Sundae();
11     Sundae x = Sundae.makeASundae();
12   }
13 } ///:~
```

private: you can't touch that! (Cont.)

- Any method that you're certain is only a "helper" method for that class can be made ***private***
- You should make all fields ***private***

protected: inheritance access

- ***Inheritance***
  - **Takes an existing class— which we refer to as the *base class*—and adds new members to that class without touching the existing class**

```
class Foo extends Bar {
```

- **The creator of the base class would like to take a particular member and grant access to derived classes**
- ***protected* also gives package access**
  - **Other classes in the same package may access *protected* elements**

protected: inheritance access (Cont.)

- **Cannot use package-access member from another package**

```
1  package access.dessert;
2
3  public class Cookie {
4    public Cookie() {
5     System.out.println("Cookie constructor");
6    }
7    void bite() { System.out.println("bite"); }
8  } ///:~
```

```
1  import access.dessert.*;
2
3  public class ChocolateChip extends Cookie {
4    public ChocolateChip() {
5     System.out.println("ChocolateChip constructor");
6    }
7    public void chomp() {
8      //! bite(); // Can't access bite
9    }
10   public static void main(String[] args) {
11     ChocolateChip x = new ChocolateChip();
12     x.chomp();
13   }
14 }
```

protected: inheritance access (Cont.)

- now *bite( )* becomes accessible to anyone inheriting from *Cookie*

```
1  package access.cookie2;
2
3  public class Cookie {
4    public Cookie() {
5      System.out.println("Cookie constructor");
6    }
7    protected void bite() {
8      System.out.println("bite");
9    }
10 } ///:~
```

```
1  import access.cookie2.*;
2
3  public class ChocolateChip2 extends Cookie {
4    public ChocolateChip2() {
5     System.out.println("ChocolateChip2 constructor");
6    }
7    public void chomp() { bite(); } // Protected method
8    public static void main(String[] args) {
9      ChocolateChip2 x = new ChocolateChip2();
10     x.chomp();
11   }
12 }
```

Summary of access specifiers

	In the same class	In the same packages	Subclass in the different packages	Non-subclass in the different packages
private	Yes			
default (Package-access)	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

Interface and implementation

- Access control is often referred to as *implementation hiding*
- Wrapping data and methods within classes in combination with implementation hiding is often called *encapsulation*
  - The result is a data type with characteristics and behaviors
- Access control puts boundaries within a data type for two important reasons
  - Establish what the client programmers can and cannot use
  - Separate the interface from the implementation

Interface and implementation (Cont.)

- For clarity, put the *public* members at the beginning, followed by the *protected*, *package-access*, and *private* members

```
1  public class OrganizedByAccess {
2    public void pub1() { /* ... */ }
3    public void pub2() { /* ... */ }
4    public void pub3() { /* ... */ }
5    private void priv1() { /* ... */ }
6    private void priv2() { /* ... */ }
7    private void priv3() { /* ... */ }
8    private int i;
9    // ...
10 } ///:~
```

Class access

- To control the access of a class, the specifier must appear before the keyword ***class***

```
public class Widget {
```

- If the name of your library is *access*, any client programmer can access *Widget* by

```
import access.Widget;
```
or
```
import access.*;
```

- An extra set of constraints:
  - Only one ***public*** class per compilation unit (file)
  - The name of the ***public*** class must exactly match the name of the file containing the compilation unit, including capitalization
  - It is possible, though not typical, to have a compilation unit with ***no public*** class at all

Class access (Cont.)

- A class cannot be *private* or *protected*
- Two choices for class access: *package access* or *public*
- What if you don't want anyone else to have access to that class
  - Make all the constructors *private*, thereby preventing anyone but you, inside a *static* member of the class, from creating an object of that class

Class access (Cont.)

```
1  class Soup1 {
2    private Soup1() {}
3    // (1) Allow creation via static method:
4    public static Soup1 makeSoup() {
5      return new Soup1();
6    }
7  }
8
9  class Soup2 {
10   private Soup2() {}
11   // (2) Create a static object and return a reference
12   // upon request.(The "Singleton" pattern):
13   private static Soup2 ps1 = new Soup2();
14   public static Soup2 access() {
15     return ps1;
16   }
17   public void f() {}
18 }
19
20 // Only one public class allowed per file:
21 public class Lunch {
22   void testPrivate() {
23     // Can't do this! Private constructor:
24     //! Soup1 soup = new Soup1();
25   }
26   void testStatic() {
27     Soup1 soup = Soup1.makeSoup();
28   }
29   void testSingleton() {
30     Soup2.access().f();
31   }
32 } ///:~
```

- Make a class effectively private with *private* constructors
- Soup2 uses what's called a *design pattern*
- This particular pattern is called a *Singleton*, because it allows only a single object to ever be created

Thank you

zhenling@seu.edu.cn