

Lecture 4 - Java Fundamentals

CS260 – Android App Development

Paul Cao

Review

- File I/O
- Arrays in Java
- methods for class

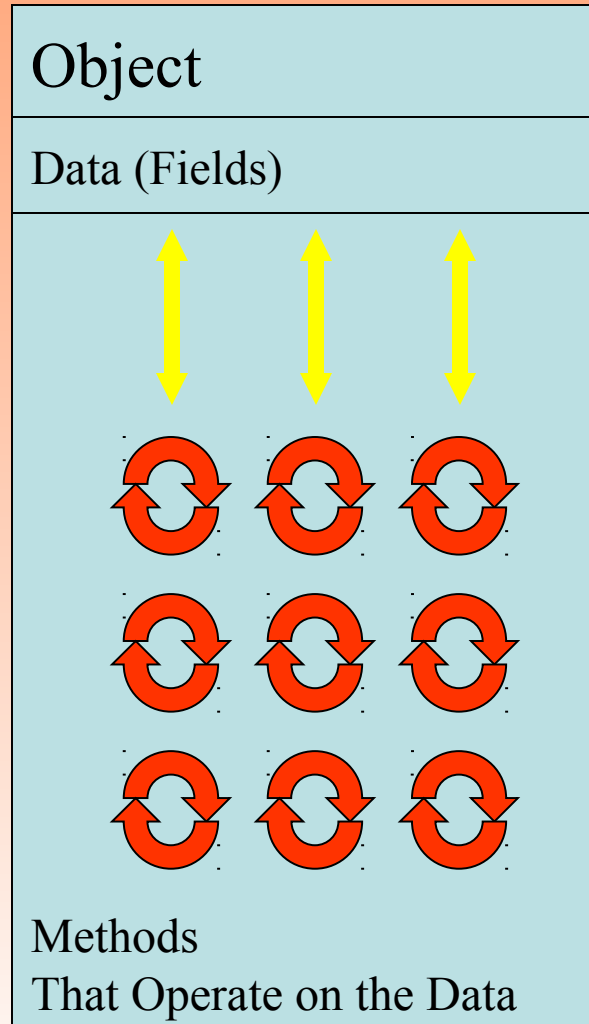
Plan for today

- Java class and objects
- Inheritance and polymorphism
- Interface
- Exception handling

Object-Oriented Programming

- Object-oriented programming is centered on creating objects rather than procedures.
- Objects are a melding of data and procedures that manipulate that data.
- Data in an object are known as *fields*.
- Procedures in an object are known as *methods*.

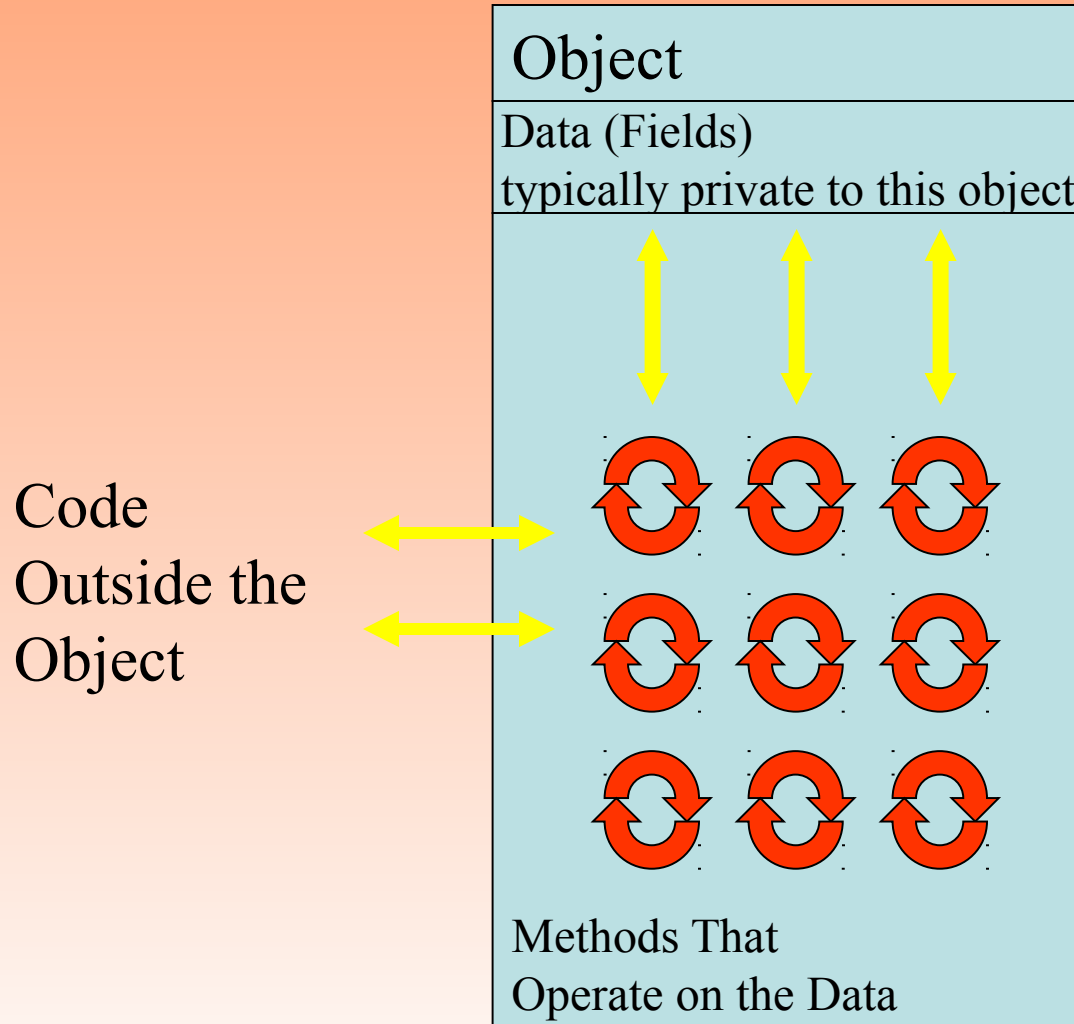
Object-Oriented Programming



Object-Oriented Programming

- Object-oriented programming combines data and behavior via *encapsulation*.
- *Data hiding* is the ability of an object to hide data from other objects in the program.
- Only an object's methods should be able to directly manipulate its data.
- Other objects are allowed manipulate an object's data via the object's methods.

Object-Oriented Programming



Object-Oriented Programming

Data Hiding

- Data hiding is important for several reasons.
 - It protects the data from accidental corruption by outside objects.
 - It hides the details of how an object works, so the programmer can concentrate on using it.
 - It allows the maintainer of the object to have the ability to modify the internal functioning of the object without “breaking” someone else's code.

Object-Oriented Programming

Code Reusability

- Object-Oriented Programming (OOP) has encouraged object reusability.
- A software object contains data and methods that represents a specific concept or service.
- An object is not a stand-alone program.
- Objects can be used by programs that need the object's service.
- Reuse of code promotes the rapid development of larger software projects.

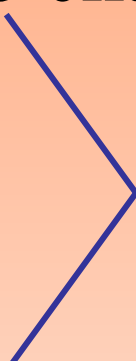
An Everyday Example of an Object— An Alarm Clock

- Fields define the state that the alarm is currently in.
 - The current second (a value in the range of 0-59)
 - The current minute (a value in the range of 0-59)
 - The current hour (a value in the range of 1-12)
 - The time the alarm is set for (a valid hour and minute)
 - Whether the alarm is on or off (“on” or “off”)


An Everyday Example of an Object— An Alarm Clock

- Methods are used to change a field's value

- Set time
- Set alarm time
- Turn alarm on
- Turn alarm off
- Increment the current second
- Increment the current minute
- Increment the current hour
- Sound alarm



Public methods are
accessed by users outside
the object.



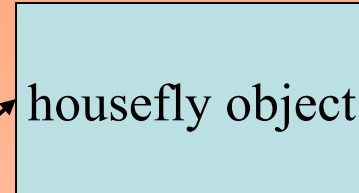
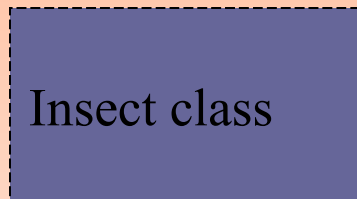
Private methods
are part of the
object's internal
design.

Classes and Objects

- The programmer determines the fields and methods needed, and then creates a class.
- A class can specify the fields and methods that a particular type of object may have.
- A class is a “blueprint” that objects may be created from.
- A class is not an object, but it can be a description of an object.
- An object created from a class is called an *instance* of the class.

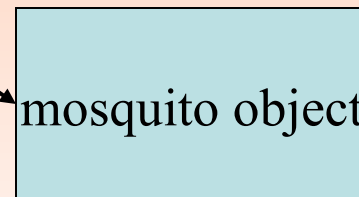
Classes and Objects

The Insect class defines the fields and methods that will exist in all objects that are an instances of the Insect class.



The housefly object is an instance of the Insect class.

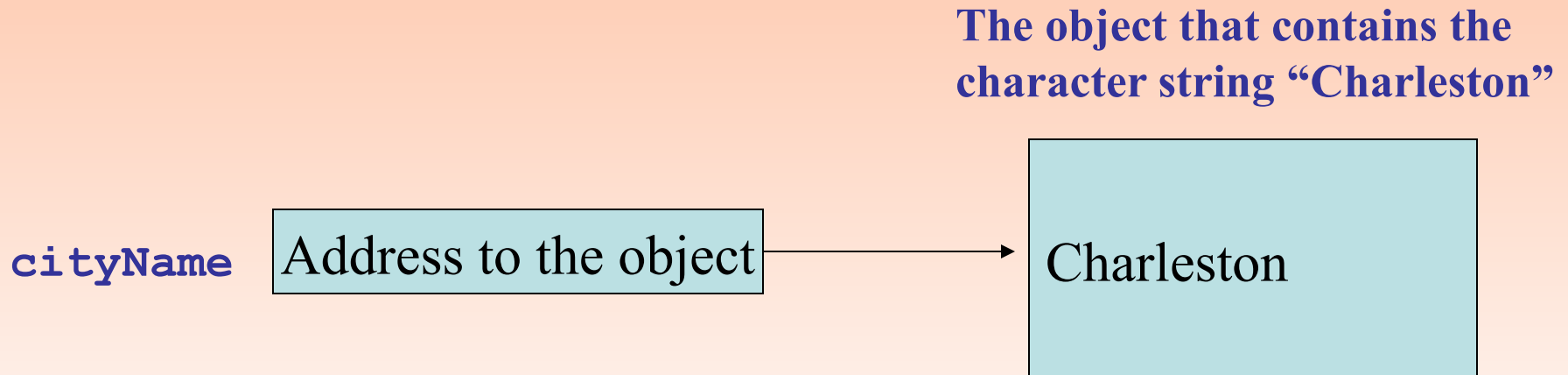
The mosquito object is an instance of the Insect class.



Classes

- We learned that a reference variable contains the address of an object.

```
String cityName = "Charleston";
```



Classes

- The `length()` method of the `String` class returns an integer value that is equal to the length of the string.

```
int stringLength = cityName.length();
```
- Class objects normally have methods that perform useful operations on their data.
- Primitive variables can only store data and have no methods.

Classes and Instances

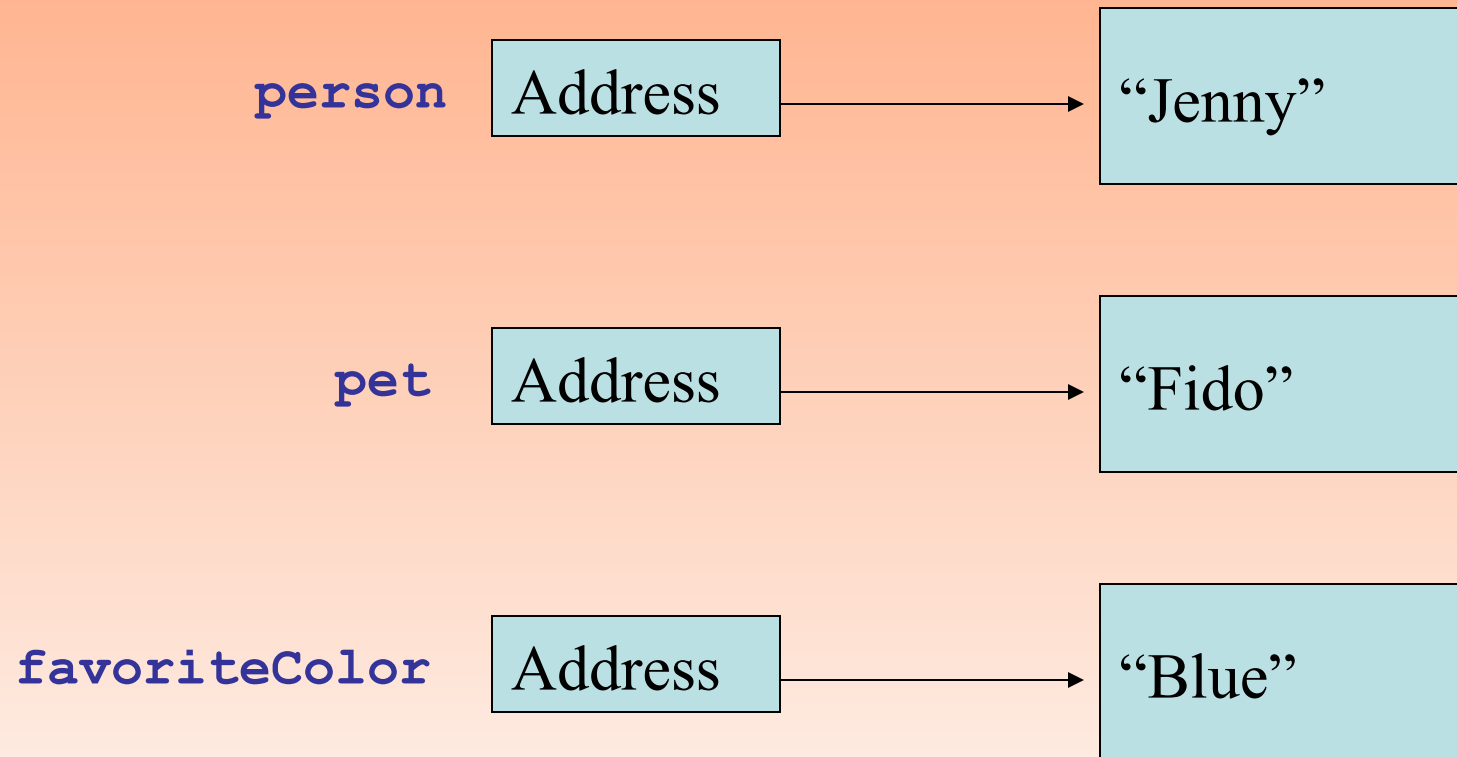
- Many objects can be created from a class.
- Each object is independent of the others.

```
String person = "Jenny ";
```

```
String pet = "Fido";
```

```
String favoriteColor = "Blue";
```


Classes and Instances



Classes and Instances

- Each instance of the `String` class contains different data.
- The instances all share the same design.
- Each instance has all of the attributes and methods that were defined in the `String` class.
- Classes are defined to represent a single concept or service.

Building a Rectangle class

- A Rectangle object will have the following fields:
 - length. The length field will hold the rectangle's length.
 - width. The width field will hold the rectangle's width.

Building a Rectangle class

- The Rectangle class will also have the following methods:
 - **setLength.** The setLength method will store a value in an object's length field.
 - **setWidth.** The setWidth method will store a value in an object's width field.
 - **getLength.** The getLength method will return the value in an object's length field.
 - **getWidth.** The getWidth method will return the value in an object's width field.
 - **getArea.** The getArea method will return the area of the rectangle, which is the result of the object's length multiplied by its width.

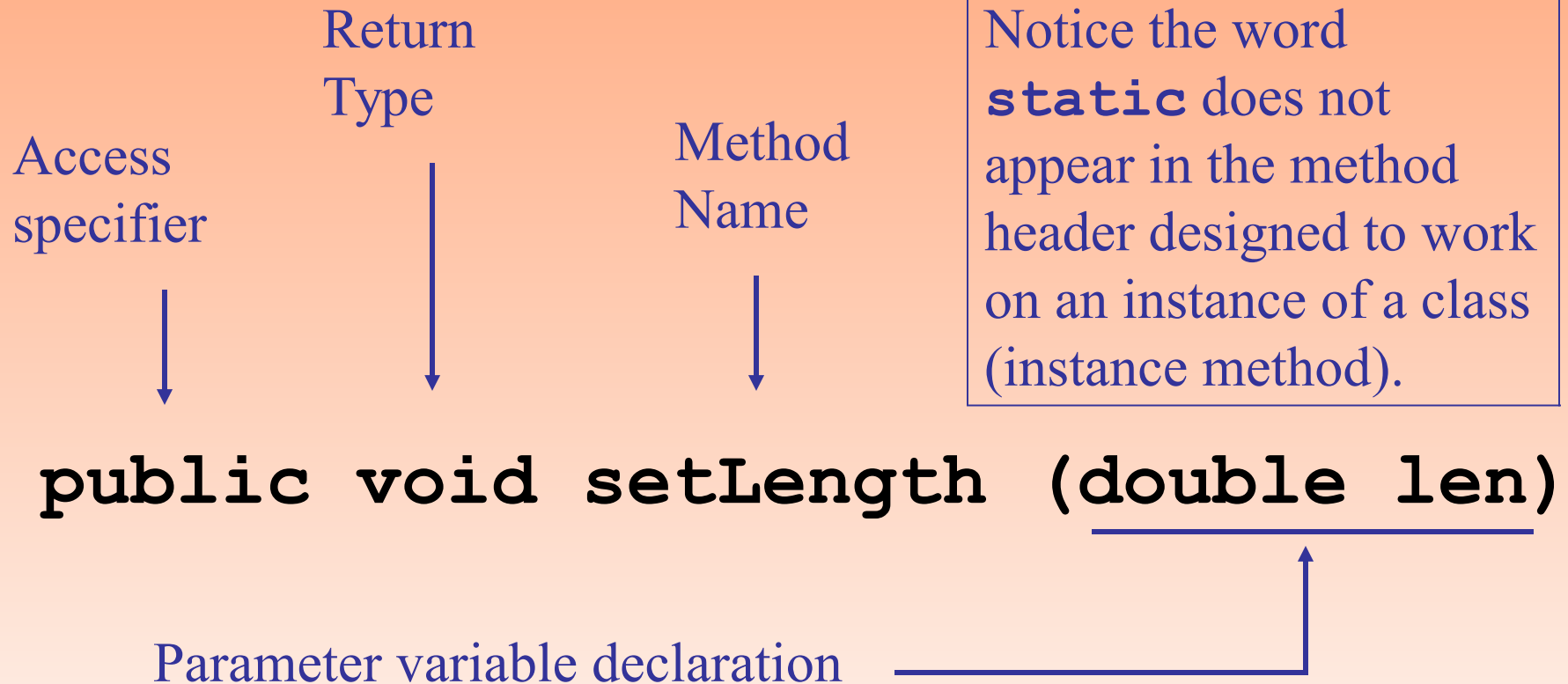
Writing the Code for the Class Fields

```
public class Rectangle
{
    private double length;
    private double width;
}
```

Access Specifiers

- An access specifier is a Java keyword that indicates how a field or method can be accessed.
- `public`
 - When the `public` access specifier is applied to a class member, the member can be accessed by code inside the class or outside.
- `private`
 - When the `private` access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.

Header for the `setLength` Method



Writing and Demonstrating the setLength Method

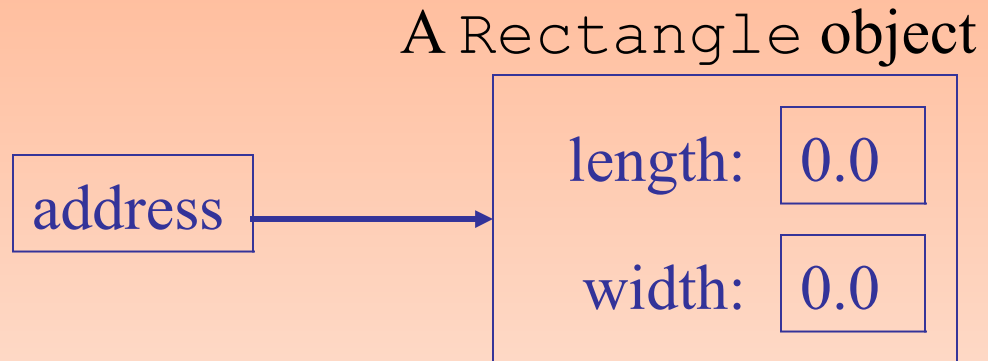
```
/**  
    The setLength method stores a value in the  
    length field.  
    @param len The value to store in length.  
*/  
public void setLength(double len)  
{  
    length = len;  
}
```

Examples: Rectangle.java, LengthDemo.java
(example 1)

Creating a Rectangle object

```
Rectangle box = new Rectangle ();
```

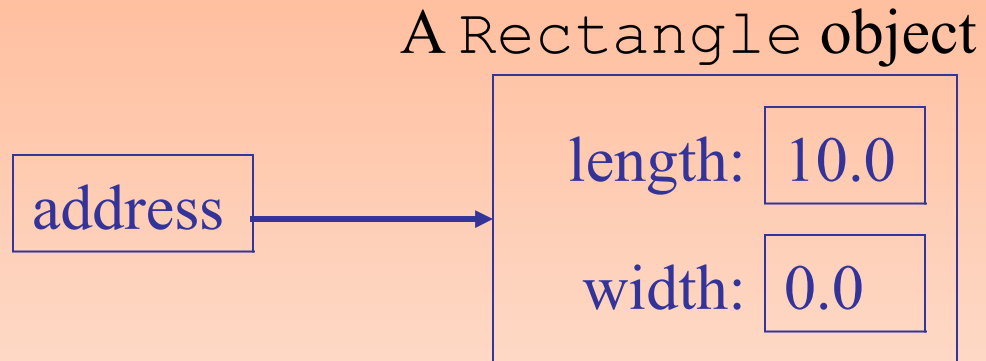
The `box` variable holds the address of the Rectangle object.



Calling the `setLength` Method

```
box.setLength(10.0);
```

The `box` variable holds the address of the `Rectangle` object.



This is the state of the `box` object after the `setLength` method executes.

Writing the `getLength` Method

```
/**  
    The getLength method returns a Rectangle  
    object's length.  
    @return The value in the length field.  
*/  
public double getLength()  
{  
    return length;  
}
```

Similarly, the `setWidth` and `getWidth` methods can be created.

Writing and Demonstrating the `getArea` Method

```
/**  
    The getArea method returns a Rectangle  
    object's area.  
    @return The product of length times width.  
*/  
public double getArea()  
{  
    return length * width;  
}
```

Examples: `Rectangle.java`, `RoomArea.java` (example 2)

Accessor and Mutator Methods

- Because of the concept of data hiding, fields in a class are private.
- The methods that retrieve the data of fields are called *accessors*.
- The methods that modify the data of fields are called *mutators*.
- Each field that the programmer wishes to be viewed by other classes needs an accessor.
- Each field that the programmer wishes to be modified by other classes needs a mutator.

Accessors and Mutators

- For the `Rectangle` example, the accessors and mutators are:
 - **setLength** : Sets the value of the `length` field.
`public void setLength(double len) ...`
 - **setWidth** : Sets the value of the `width` field.
`public void setLength(double w) ...`
 - **getLength** : Returns the value of the `length` field.
`public double getLength() ...`
 - **getWidth** : Returns the value of the `width` field.
`public double getWidth() ...`
- Other names for these methods are *getters* and *setters*.

Stale Data

- Some data is the result of a calculation.
- Consider the area of a rectangle.
 - $length \times width$
- It would be impractical to use an *area* variable here.
- Data that requires the calculation of various factors has the potential to become *stale*.
- To avoid stale data, it is best to calculate the value of that data within a method rather than store it in a variable.

Stale Data

- Rather than use an area variable in a Rectangle class:

```
public double getArea()  
{  
    return length * width;  
}
```

- This dynamically calculates the value of the rectangle's area when the method is called.
- Now, any change to the `length` or `width` variables will not leave the area of the rectangle stale.

Class Layout Conventions

- The layout of a source code file can vary by employer or instructor.
- A common layout is:
 - Fields listed first
 - Methods listed second
 - Accessors and mutators are typically grouped.
- There are tools that can help in formatting layout to specific standards.

Instance Fields and Methods

- Fields and methods that are declared as previously shown are called *instance fields* and *instance methods*.
- Objects created from a class each have their own copy of instance fields.
- Instance methods are methods that are not declared with a special keyword, `static`.

Instance Fields and Methods

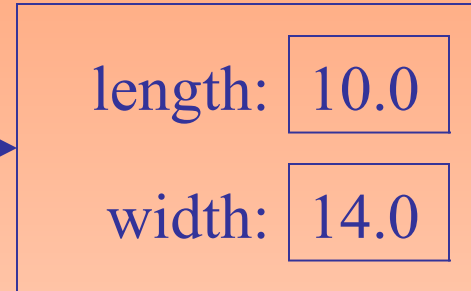
- Instance fields and instance methods require an object to be created in order to be used.
- Note that each room represented in this example can have different dimensions.

```
Rectangle kitchen = new Rectangle();  
Rectangle bedroom = new Rectangle();  
Rectangle den = new Rectangle();
```

States of Three Different Rectangle Objects

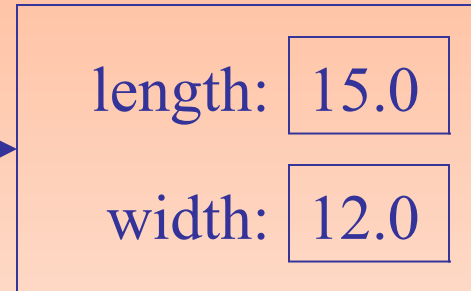
The kitchen variable holds the address of a Rectangle Object.

address



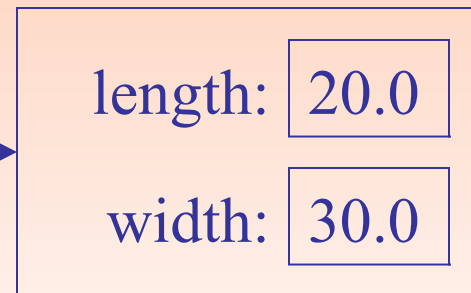
The bedroom variable holds the address of a Rectangle Object.

address



The den variable holds the address of a Rectangle Object.

address



Constructors

- Classes can have special methods called *constructors*.
- A constructor is a method that is automatically called when an object is created.
- Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.

Constructors

- Constructors have a few special properties that set them apart from normal methods.
 - Constructors have the same name as the class.
 - Constructors have no return type (not even `void`).
 - Constructors may not return any values.
 - Constructors are typically public.

Constructor for Rectangle Class

```
/**
    Constructor
    @param len The length of the rectangle.
    @param w The width of the rectangle.
 */
public Rectangle(double len, double w)
{
    length = len;
    width = w;
}
```

Uninitialized Local Reference Variables

- Reference variables can be declared without being initialized.

```
Rectangle box;
```

- This statement does not create a `Rectangle` object, so it is an uninitialized local reference variable.
- A local reference variable must reference an object before it can be used, otherwise a compiler error will occur.

```
box = new Rectangle(7.0, 14.0);
```

- `box` will now reference a `Rectangle` object of length 7.0 and width 14.0.

The Default Constructor

- When an object is created, its constructor is always called.
- If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*.
 - It sets all of the object's numeric fields to 0.
 - It sets all of the object's `boolean` fields to `false`.
 - It sets all of the object's reference variables to the special value *null*.

The Default Constructor

- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The only time that Java provides a default constructor is when you do not write any constructor for a class.
- A default constructor is not provided by Java if a constructor is already written.
 - See example: Rectangle.java with no default constructor (example 3)

The `String` Class Constructor

- One of the `String` class constructors accepts a string literal as an argument.
- This string literal is used to initialize a `String` object.
- For instance:

```
String name = new String("Michael Long");
```

The `String` Class Constructor

- This creates a new reference variable *name* that points to a `String` object that represents the name “Michael Long”
- Because they are used so often, `String` objects can be created with a shorthand:

```
String name = "Michael Long";
```

Overloading Methods and Constructors

- Two or more methods in a class may have the same name as long as their parameter lists are different.
- When this occurs, it is called *method overloading*. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.

Overloaded Method add

```
public int add(int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

```
public String add (String str1, String str2)
{
    String combined = str1 + str2;
    return combined;
}
```

Rectangle Class Constructor Overload

If we were to add the no-arg constructor we wrote previously to our `Rectangle` class in addition to the original constructor we wrote, what would happen when we execute the following calls?

```
Rectangle box1 = new Rectangle();  
Rectangle box2 = new Rectangle(5.0, 10.0);
```

The BankAccount Exercise

BankAccount
-balance:double
+BankAccount() +BankAccount(startBalance:double) +BankAccount(startBalance:String) +deposit(amount:double):void +deposit(str:String):void +withdraw(amount:double):void +withdraw(str:String):void +setBalance(b:double):void +setBalance(str:String):void +getBalance():double +toString():String

Scope of Instance Fields

- Variables declared as instance fields in a class can be accessed by any instance method in the same class as the field.
- If an instance field is declared with the `public` access specifier, it can also be accessed by code outside the class, as long as an instance of the class exists.

Packages and `import` Statements

- Classes in the Java API are organized into *packages*.
- Explicit and Wildcard `import` statements
 - Explicit imports name a specific class
 - `import java.util.Scanner;`
 - Wildcard imports name a package, followed by an `*`
 - `import java.util.*;`
- The `java.lang` package is automatically made available to any Java class.

Other notes on Java classes

- Static fields and methods
- toString(), equals(), this, garbage collection
- Inheritance
- Polymorphism

Static Class Members

- *Static fields* and *static methods* do not belong to a single instance of a class.
- To invoke a static method or use a static field, the class name, rather than the instance name, is used.
- Example:

```
double val = Math.sqrt(25.0);
```



Static Fields

- Class fields are declared using the `static` keyword between the access specifier and the field type.
`private static int instanceCount = 0;`
- The field is initialized to 0 only once, regardless of the number of times the class is instantiated.
 - Primitive static fields are initialized to 0 if no initialization is performed.
- Examples: `Countable.java`, `StaticDemo.java` (Example 4)

Static Methods

- Methods can also be declared static by placing the `static` keyword between the access modifier and the return type of the method.

```
public static double milesToKilometers(double miles)
{...}
```

- When a class contains a static method, it is not necessary to create an instance of the class in order to use the method.

```
double kilosPerMile = Metric.milesToKilometers(1.0);
```

- Examples: [Metric.java](#), [MetricDemo.java](#) (Example 5)

Static Methods

- Static methods are convenient because they may be called at the class level.
- They are typically used to create utility classes, such as the `Math` class in the Java Standard Library.
- Static methods may not communicate with instance fields, only static fields.

The toString Method

- The toString method of a class can be called *explicitly*:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);  
System.out.println(xyzCompany.toString());
```

- However, the toString method does not have to be called explicitly but is called implicitly whenever you pass an object of the class to println or print.

```
Stock xyzCompany = new Stock ("XYZ", 9.62);  
System.out.println(xyzCompany);
```


The toString method

- The toString method is also called implicitly whenever you concatenate an object of the class with a string.

```
Stock xyzCompany = new Stock ("XYZ", 9.62);  
System.out.println("The stock data is:\n" +  
    xyzCompany);
```

The `toString` Method

- All objects have a `toString` method that returns the class name and a hash of the memory address of the object.
- We can override the default method with our own to print out more useful information.

The `equals` Method

- When the `==` operator is used with reference variables, the memory address of the objects are compared.
- The contents of the objects are not compared.
- All objects have an `equals` method.
- The default operation of the `equals` method is to compare memory addresses of the objects (just like the `==` operator).

The equals Method

- The Stock class has an equals method.
- If we try the following:

```
Stock stock1 = new Stock("GMX", 55.3);
Stock stock2 = new Stock("GMX", 55.3);
if (stock1 == stock2) // This is a mistake.
    System.out.println("The objects are the same.");
else
    System.out.println("The objects are not the same.");
```

only the addresses of the objects are compared.

The equals Method

- Instead of using the `==` operator to compare two `Stock` objects, we should use the `equals` method.

```
public boolean equals(Stock object2)
{
    boolean status;

    if(symbol.equals(Object2.symbol && sharePrice == Object2.sharePrice)
        status = true;
    else
        status = false;
    return status;
}
```

- Now, objects can be compared by their contents rather than by their memory addresses.
- See example: `StockCompare.java`

The `this` Reference

- The `this` reference is simply a name that an object can use to refer to itself.
- The `this` reference can be used to overcome shadowing and allow a parameter to have the same name as an instance field.

```
public void setFeet(int feet)
{
    this.feet = feet;
    //sets the this instance's feet field
    //equal to the parameter feet.
}
```

Local parameter variable feet

Shadowed instance variable

The `this` Reference

- The `this` reference can be used to call a constructor from another constructor.

```
public Stock(String sym)
{
    this(sym, 0.0);
}
```

- This constructor would allow an instance of the `Stock` class to be created using only the symbol name as a parameter.
 - It calls the constructor that takes the symbol and the price, using *sym* as the symbol argument and 0 as the price argument.
- Elaborate constructor chaining can be created using this technique.
- If `this` is used in a constructor, it must be the first statement in the constructor.

Garbage Collection

- When objects are no longer needed they should be destroyed.
- This frees up the memory that they consumed.
- Java handles all of the memory operations for you.
- Simply set the reference to *null* and Java will reclaim the memory.

The `finalize` Method

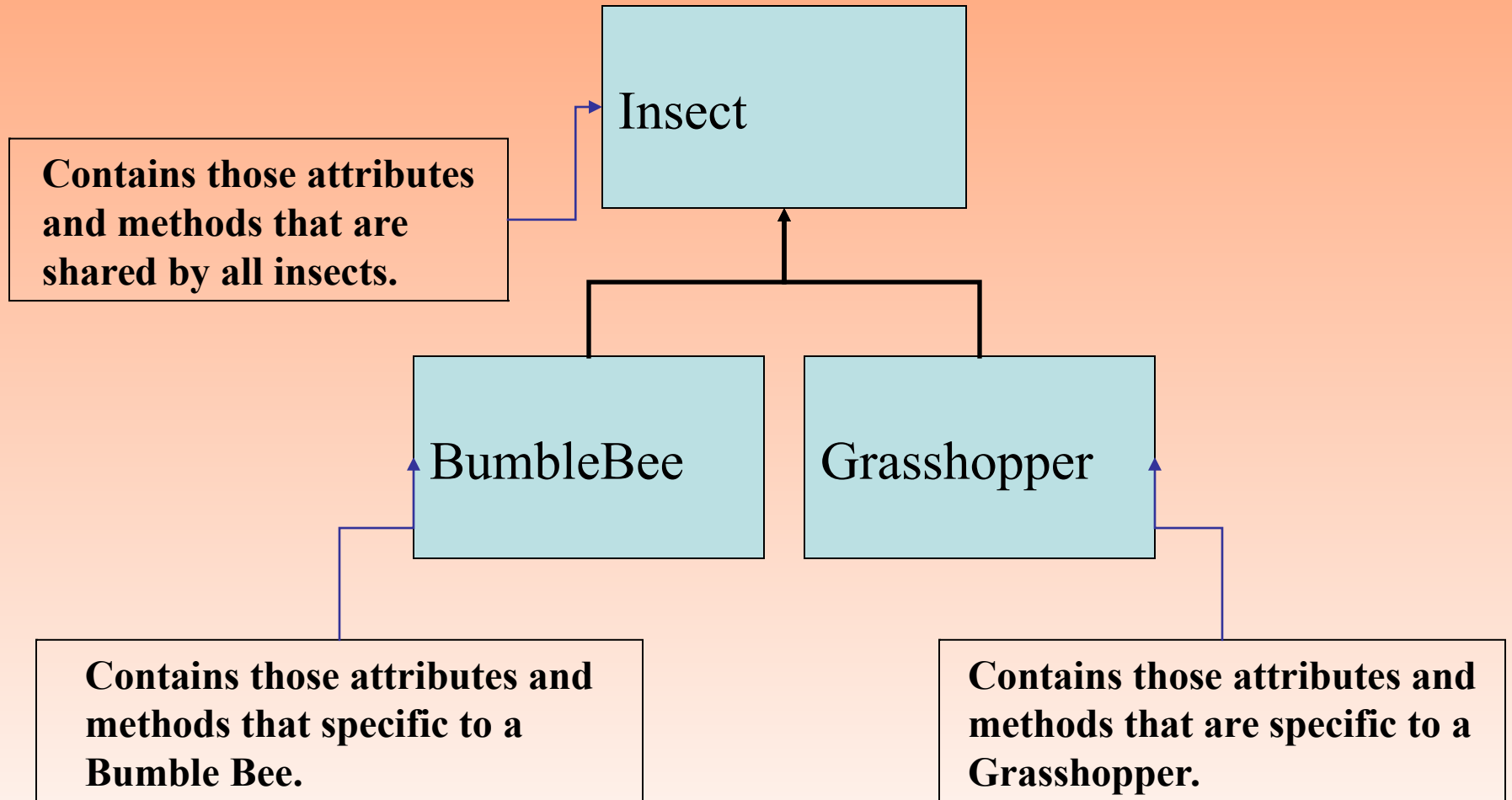
- If a method with the signature:

```
public void finalize() {...}
```

is included in a class, it will run just prior to the garbage collector reclaiming its memory.

- The garbage collector is a background thread that runs periodically.
- It cannot be determined when the `finalize` method will actually be run.

Inheritance



The “is a” Relationship

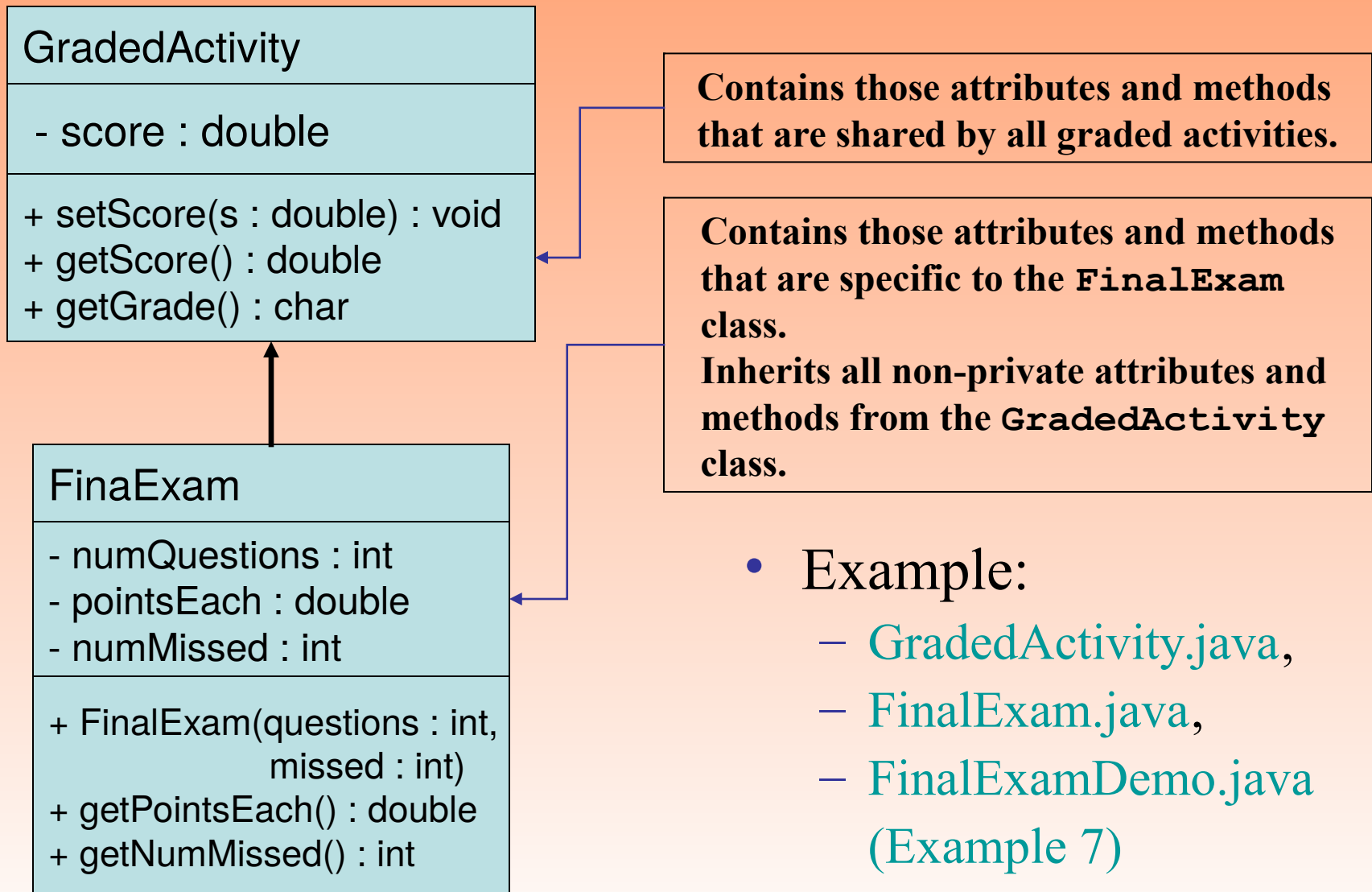
- We can *extend* the capabilities of a class.
- Inheritance involves a superclass and a subclass.
 - The *superclass* is the general class and
 - the *subclass* is the specialized class.
- The subclass is based on, or extended from, the superclass.
 - Superclasses are also called *base classes*, and
 - subclasses are also called *derived classes*.
- The relationship of classes can be thought of as *parent classes* and *child classes*.

Inheritance

- The subclass inherits fields and methods from the superclass without any of them being rewritten.
- New fields and methods may be added to the subclass.
- The Java keyword, *extends*, is used on the class header to define the subclass.

```
public class FinalExam extends GradedActivity
```

The GradedActivity Example



Inheritance, Fields and Methods

- Members of the superclass that are marked *private*:
 - are not inherited by the subclass,
 - exist in memory when the object of the subclass is created
 - may only be accessed from the subclass by public methods of the superclass.
- Members of the superclass that are marked *public*:
 - are inherited by the subclass, and
 - may be directly accessed from the subclass.

Inheritance and Constructors

- Constructors are not inherited.
- When a subclass is instantiated, the superclass default constructor is executed first.
- The `super` keyword refers to an object's superclass.
- The superclass constructor can be explicitly called from the subclass by using the `super` keyword.

[Rectangle.java](#), [Cube.java](#), [CubeDemo.java](#) (Example 8)

Overriding Superclass Methods

- A subclass may have a method with the same signature as a superclass method.
- The subclass method overrides the superclass method.
- This is known as *method overriding*.

Overriding Superclass Methods

- Recall that a method's *signature* consists of:
 - the method's name
 - the data types method's parameters in the order that they appear.
- A subclass method that overrides a superclass method must have the same signature as the superclass method.
- An object of the subclass invokes the subclass's version of the method, not the superclass's.

Overriding Superclass Methods

- An subclass method can call the overridden superclass method via the `super` keyword.

```
super.setScore(rawScore * percentage) ;
```

- There is a distinction between overloading a method and overriding a method.
- Overloading is when a method has the same name as one or more other methods, but with a different signature.
- When a method overrides another method, however, they both have the same signature.

Overriding Superclass Methods

- Both overloading and overriding can take place in an inheritance relationship.
- Overriding can only take place in an inheritance relationship.
- Example 9:
 - `SuperClass3.java`,
 - `SubClass3.java`,
 - `ShowValueDemo.java`

The Object Class

- All Java classes are directly or indirectly derived from a class named `Object`.
- `Object` is in the `java.lang` package.
- Any class that does not specify the `extends` keyword is automatically derived from the `Object` class.

```
public class MyClass
{
    // This class is derived from Object.
}
```

- Ultimately, every class is derived from the `Object` class.

The Object Class

- Because every class is directly or indirectly derived from the `Object` class:
 - every class inherits the `Object` class's members.
 - example: `toString` and `equals`.
- In the `Object` class, the `toString` method returns a string containing the object's class name and a hash of its memory address.
- The `equals` method accepts the address of an object as its argument and returns `true` if it is the same as the calling object's address.

Exercise

Write a class `Auto` with the following data

- Number of doors
- Type of transmission (manual or automatic)
- Provide necessary constructors, access and mutate functions, and a `toString()`, a `equals()`

Write a class `Truck` that inherits from `Auto`

- Load (how many tons it can tow)
- Provide necessary constructors, access and mutate, a `toString()`, a `equals()`

Write a driver code to test the two classes.

Polymorphism

- A reference variable can reference objects of classes that are derived from the variable's class.

```
GradedActivity exam;
```

- We can use the exam variable to reference a `GradedActivity` object.

```
exam = new GradedActivity();
```

- The `GradedActivity` class is also used as the superclass for the `FinalExam` class.
- An object of the `FinalExam` class *is a* `GradedActivity` object.

Polymorphism

- A `GradedActivity` variable can be used to reference a `FinalExam` object.

```
GradedActivity exam = new FinalExam(50, 7);
```
- This statement creates a `FinalExam` object and stores the object's address in the `exam` variable.
- This is an example of polymorphism.
- The term *polymorphism* means the ability to take many forms.
- In Java, a reference variable is *polymorphic* because it can reference objects of types different from its own, as long as those types are subclasses of its type.

Polymorphism

- Other legal polymorphic references:

```
GradedActivity exam1 = new FinalExam(50, 7);  
GradedActivity exam2 = new PassFailActivity(70);  
GradedActivity exam3 = new PassFailExam(100, 10, 70);
```

- The GradedActivity class has three methods: setScore, getScore, and getGrade.
- A GradedActivity variable can be used to call only those three methods.

```
GradedActivity exam = new PassFailExam(100, 10, 70);  
System.out.println(exam.getScore()); // This works.  
System.out.println(exam.getGrade()); // This works.  
System.out.println(exam.getPointsEach()); // ERROR!
```

Polymorphism and Dynamic Binding

- If the object of the subclass has overridden a method in the superclass:
 - If the variable makes a call to that method the subclass's version of the method will be run.

```
GradedActivity exam = new PassFailActivity(60);  
exam.setScore(70);  
System.out.println(exam.getGrade());
```

- Java performs *dynamic binding* or *late binding* when a variable contains a polymorphic reference.
- The Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references.

Polymorphism

- It is the object's type, rather than the reference type, that determines which method is called.
- Example:
 - Polymorphic.java
- You cannot assign a superclass object to a subclass reference variable.

Abstract Classes

- An abstract class cannot be instantiated, but other classes are derived from it.
- An *Abstract class* serves as a superclass for other classes.
- The abstract class represents the generic or abstract form of all the classes that are derived from it.
- A class becomes abstract when you place the abstract key word in the class definition.

```
public abstract class ClassName
```

Abstract Methods

- An abstract method has no body and must be overridden in a subclass.
- An *abstract method* is a method that appears in a superclass, but expects to be overridden in a subclass.
- An abstract method has only a header and no body.

`AccessSpecifier abstract ReturnType MethodName (ParameterList) ;`

- Example:
 - Student.java, CompSciStudent.java, CompSciStudentDemo.java

Abstract Methods

- Notice that the key word `abstract` appears in the header, and that the header ends with a semicolon.

```
public abstract void setValue(int value) ;
```

- Any class that contains an abstract method is automatically abstract.
- If a subclass fails to override an abstract method, a compiler error will result.
- Abstract methods are used to ensure that a subclass implements the method.

Interfaces

- An *interface* is similar to an abstract class that has all abstract methods.
 - It cannot be instantiated, and
 - all of the methods listed in an interface must be written elsewhere.
- The purpose of an interface is to specify behavior for other classes.
- An interface looks similar to a class, except:
 - the keyword `interface` is used instead of the keyword `class`, and
 - the methods that are specified in an interface have no bodies, only headers that are terminated by semicolons.

Interfaces

- The general format of an interface definition:

```
public interface InterfaceName
{
    (Method headers...)
}
```

- All methods specified by an interface are public by default.
- A class can implement one or more interfaces.

Interfaces

- If a class implements an interface, it uses the `implements` keyword in the class header.

```
public class FinalExam3 extends GradedActivity  
    implements Relatable
```

- Example 10:
 - GradedActivity.java
 - Relatable.java
 - FinalExam3.java
 - InterfaceDemo.java

Fields in Interfaces

- An interface can contain field declarations:
 - all fields in an interface are treated as `final` and `static`.
- Because they automatically become `final`, you must provide an initialization value.

```
public interface Doable
{
    int FIELD1 = 1, FIELD2 = 2;
    (Method headers...)
}
```

- In this interface, `FIELD1` and `FIELD2` are `final static int` variables.
- Any class that implements this interface has access to these variables.

Implementing Multiple Interfaces

- A class can be derived from only one superclass.
- Java allows a class to implement multiple interfaces.
- When a class implements multiple interfaces, it must provide the methods specified by all of them.
- To specify multiple interfaces in a class definition, simply list the names of the interfaces, separated by commas, after the implements key word.

```
public class MyClass implements Interface1,  
                                Interface2,  
                                Interface3
```