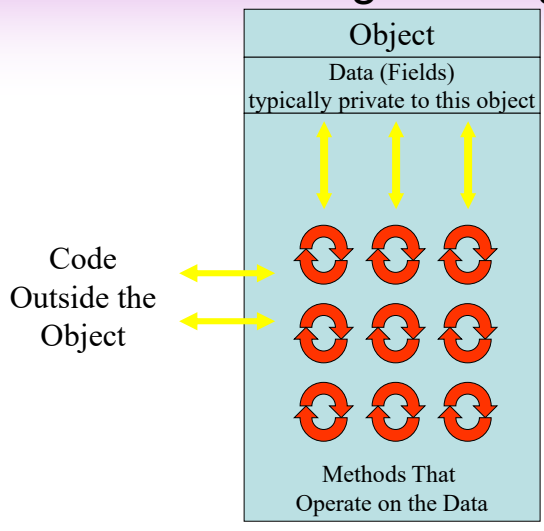
CSIT 254 - Data Structures
Classes a.k.a. Java Review

Classes

by Tony Gaddis (3rd, 4th, 5th editions)
Mucked up by Brower

Object-Oriented Programming

Recap



2

Object-Oriented Programming

Recap

- 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.
- 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.

3

Object-Oriented Programming
Code Reusability

Recap

- 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.

4

Classes and Instances

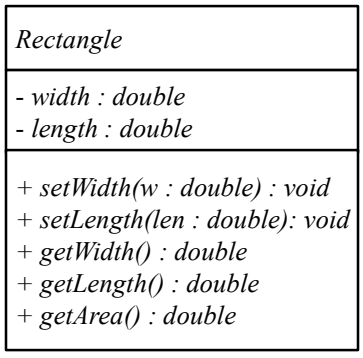
Recap

- Each instance of the `String` class contains different data.
- The instances are 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.

5

Converting the UML Diagram to Code

This is a portion of Rectangle

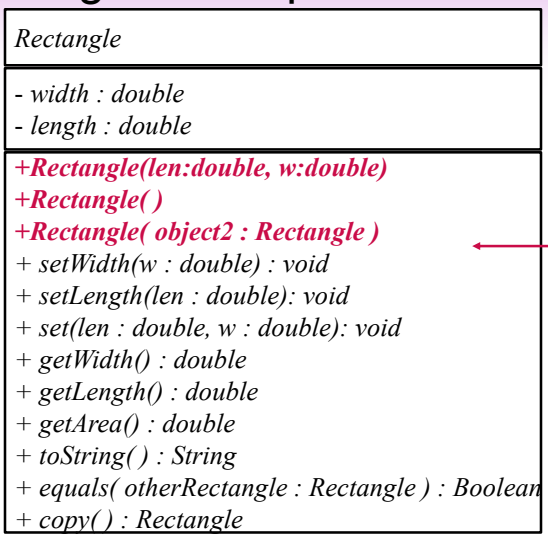



```
public class Rectangle
{
   private double width;
   private double length;

   public void setWidth(double w)
   {    width = w;
   }
   public void setLength(double len)
   {    length = len;
   }
   public double getWidth()
   {    return width;
   }
   public double getLength()
   {    return length;
   }
   public double getArea()
   {    return length * width;
   }
}
```


6

Rectangle - Complete UML



Notice there is no return type listed for constructors.

 7

States of Three Different Rectangle Objects

Recap

The `kitchen` variable holds the address of a `Rectangle` Object.

address → length: 10.0, width: 14.0

The `bedroom` variable holds the address of a `Rectangle` Object.

address → length: 15.0, width: 12.0

The `den` variable holds the address of a `Rectangle` Object.

address → length: 20.0, width: 30.0

RoomAreas.java

 8

Recap

`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 first call would use the no-arg constructor and `box1` would have a length of 1.0 and width of 1.0.

The second call would use the original constructor and `box2` would have a length of 5.0 and a width of 10.0.

RectangleDemo.java

9

What is Inheritance?

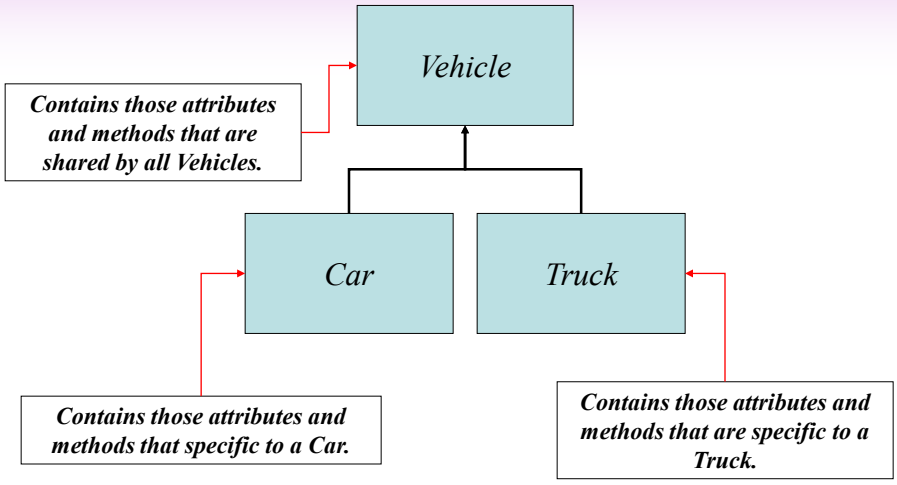
Generalization vs. Specialization

- Real-life objects are typically specialized versions of other more general objects.
- The term "Vehicle" describes a very general type of automobile with numerous characteristics.
- Cars and Trucks are Vehicles
  - They share the general characteristics of a Vehicle.
  - However, they have special characteristics of their own.
    - Trucks have large cargo areas
    - Cars have multiple doors (humor me).
- Cars and Trucks are specialized versions of a Vehicle.

10

See Vehicle.java, Car.java, Truck.java and DemoCarTruck.java

Inheritance



11

The "is a" Relationship

- The relationship between a superclass and an inherited class is called an "is a" relationship.
  - A grasshopper "is a" insect.
  - A poodle "is a" dog.
  - A car "is a" vehicle.
- A specialized object has:
  - all of the characteristics of the general object, plus
  - additional characteristics that make it special.
- In object-oriented programming, *inheritance* is used to create an "is a" relationship among classes.

12

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*.

13

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 Car extends Vehicle
```


14

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.

15

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);
```

Class name *Static method*

16

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

17

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.

18

Overloaded Methods

- Two or more methods in a class may have the same name; however, their parameter lists must be different.

```
public class MyMath{
  public static int square(int number){
    return number * number;
  }
  public static double square(double number){
    return number * number;
  }
}
```

- Example: OverloadingDemo.java

See also DisplayNeat.java and OverloadingDemo2.java and OverloadingDemo3.java

19

Overloaded Methods

- Java uses the method signature (name, type of parameters and order of parameters) to determine which method to call.
- This process is known as *binding*.
- The return type of the method is not part of the method signature.
- Example: Pay.java, WeeklyPay.java

20

The `toString` Method

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

```
Vehicle sampleVehicle = new Vehicle("silly",1972);
System.out.println(sampleVehicle.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`.

```
Car sampleCar = new Car("Chevy Corvette",2014,2);
System.out.println(sampleCar);
```

See DemoCarTruck.java

21

The `toString` method

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

```
Truck sampleTruck = new Truck("U-Haul Box Truck",2013,1611);
System.out.println(sampleTruck + " it's big!");
```


22

The `toString` Method

- All objects have a `toString` method that returns the class name and a hash of the memory address of the object.
  `The box for the pizza is: Rectangle@276af2`
- We can override the default method with our own to print out more useful information.
- Examples: Pizza.java and PizzaDemo.java

```
public String toString()
{
    return "Rectangle L: " + length + " W: " + width;
}
```


23

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).

24

The `equals` Method

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

```
Rectangle rectangle1 = new Rectangle(3.0, 4.0);
Rectangle rectangle2 = new Rectangle(3.0, 4.0);
if (rectangle1 == rectangle2) // 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

25

The `equals` Method

- Compare objects by their contents rather than by their memory addresses.
- Instead of simply using the == operator to compare two `Rectangle` objects, we should use the `equals` method.

```
public boolean equals(Rectangle otherRectangle)
{
    boolean alike;

    if (length == otherRectangle.getLength()
            && width == otherRectangle.getWidth())
        alike = true;
    else
        alike = false;

    return alike;
}
```

- See examples: RectangleCompare.java, PizzaCompare.java

26

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.

27

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.
- Example: equals( ) in Rectangle.java

28

Methods That Copy Objects

- There are two ways to copy an object.
  - You cannot use the assignment operator to copy reference types
  - Reference only copy
    - This is simply copying the address of an object into another reference variable.
  - Deep copy (correct)
    - This involves creating a new instance of the class and copying the values from one object into the new object.
  - Example: ObjectNotCopy.java then ObjectCopy.java

29

Copy Constructors

- A copy constructor accepts an existing object of the same class and clones it.

```
public Rectangle(Rectangle object2)
{
   length = object2.getLength();
   width = object2.getWidth();
}

// Create a Rectangle object
Rectangle rectangle1 = new Rectangle(8.5, 11.0);
//Create rectangle2, a copy of rectangle1
Rectangle rectangle2 = new Rectangle(rectangle1);
```

See ObjectCopy2.java

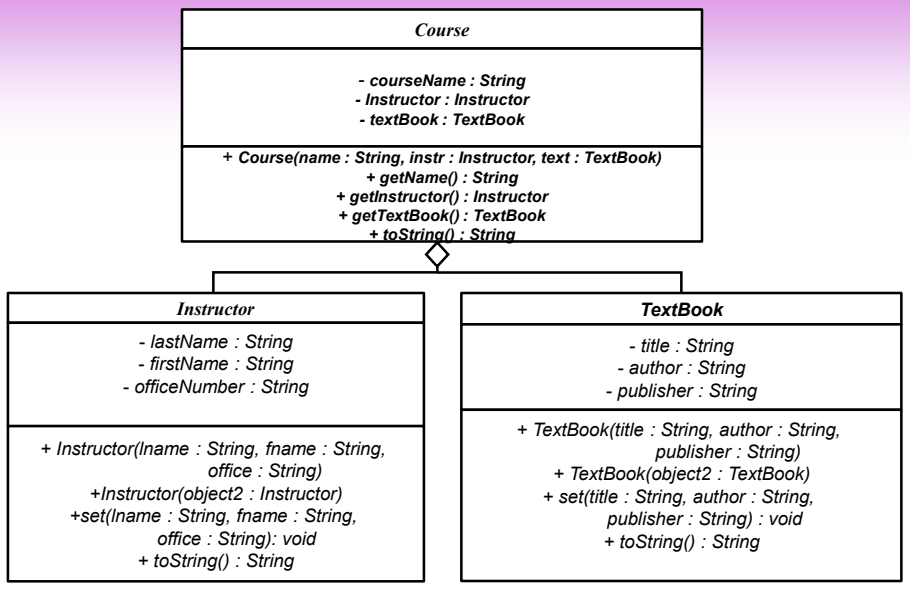
30

Aggregation

- Creating an instance of one class as a reference in another class is called *object aggregation*.
- Aggregation creates a "has a" relationship between objects.
- Examples:
  - Instructor.java, Textbook.java, Course.java, CourseDemo.java

31

Aggregation in UML Diagrams



32

Returning References to Private Fields

- Avoid returning references to private data elements.
- Returning references to private variables will allow any object that receives the reference to modify the variable.

33

Null References

- A *null reference* is a reference variable that points to nothing.
- If a reference is null, then no operations can be performed on it.
- References can be tested to see if they point to null prior to being used.

```
if(name != null)
   System.out.println("Name is: " +
                      name.toUpperCase());
```

- Examples: FullName.java, NameTester.java, NameTester2.java

34

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.

35

Garbage Collection

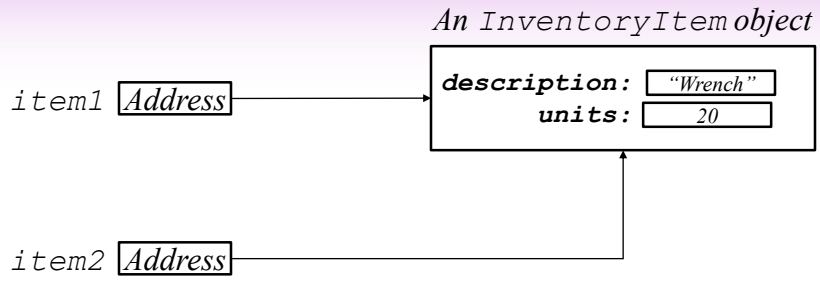
- The Java Virtual Machine has a process that runs in the background that reclaims memory from released objects.
- The *garbage collector* will reclaim memory from any object that no longer has a valid reference pointing to it.

```
InventoryItem item1 = new InventoryItem ("Wrench", 20);
InventoryItem item2 = item1;
```

- This sets item1 and item2 to point to the same object.

36

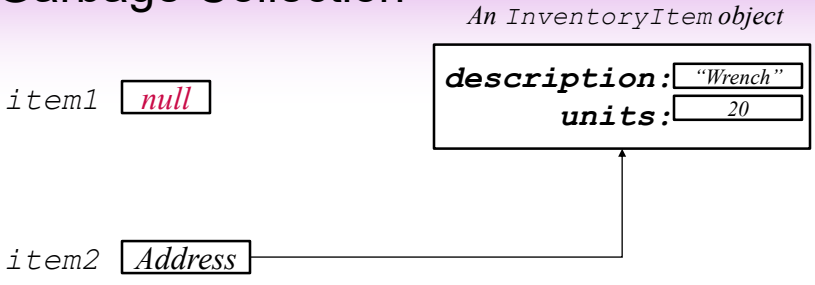
Garbage Collection



Here, both `item1` and `item2` point to the same instance of the `InventoryItem` class.

37

Garbage Collection



However, by running the command:

```
item1 = null;
```

only `item2` will be pointing to the object.

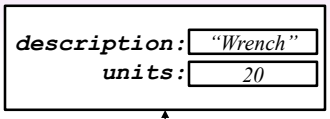
38

Garbage Collection

item1 null

item2 null

An `InventoryItem` *object*



Since there are no valid references to this object, it is now available for the garbage collector to reclaim.

If we now run the command:

```
item2 = null;
```

neither `item1` *or* `item2` *will be pointing to the object.*

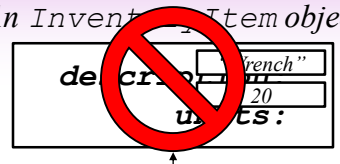
39

Garbage Collection

item1 null

item2 null

An `InventoryItem` *object*



The garbage collector reclaims the memory the next time it runs in the background.

40

Polymorphism

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

```
Vehicle sampleVehicle
```

- We can use the sampleVehicle variable to reference a `Vehicle` object.

```
sampleVehicle = new Vehicle("silly",1972);
```

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

41

Polymorphism

- A `Vehicle` variable can be used to reference a `Car` object.

```
Vehicle sampleCar = new Car("Chevy Corvette",2014,2);
```

- This statement creates a `Car` object and stores the object's address in the sampleCar Vehicle object 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.

42

Polymorphism

- Other legal polymorphic reference:

```
Vehicle sampleTruck = new Truck("U-Haul Box Truck",2013,1611);
```

- The `Vehicle` class has methods: `setYearModel`, `getYearModel`, `setMake`, and `getMake`.
- A `Vehicle` variable can be used to call only those methods.

```
Vehicle sampleCar = new Car("Chevy Corvette",2014,2);
System.out.println(sampleCar.getMake()); // This works.
System.out.println(sampleCar.getYearModel()); // This works.
System.out.println(sampleCar.getNumDoors()); // ERROR!
```

- ***However this will work:***

```
if (sampleCar instanceof Car)
   System.out.println("\n'sampleCar' is a Car - doors "
              + ((Car) sampleCar).getNumDoors());
```


43

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
```


44

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);
```


45

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.

46

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.

47

Pizza UML

PizzaDemo.java references Pizza.java

Pizza
- size : int *- numToppings : int* *- toppingDescription : String*
+Pizza (newSize : int, *newNumToppings : int,* *newToppingDescription : String)* *+ setSize(newSize : int)* *+ setNumToppings (newNumToppings : int)* *+ setToppingDescription (newToppingDescription : String)* *+ getSize ( ) : int* *+ getNumToppings ( ) : int* *+ getToppingDescription ( ) : String* *+ toString( ) : String* *+ cost() : double*

48