CENG 1004
Introduction to Object Oriented Programming

Spring 2016

WEEK 5

Today’s Topics

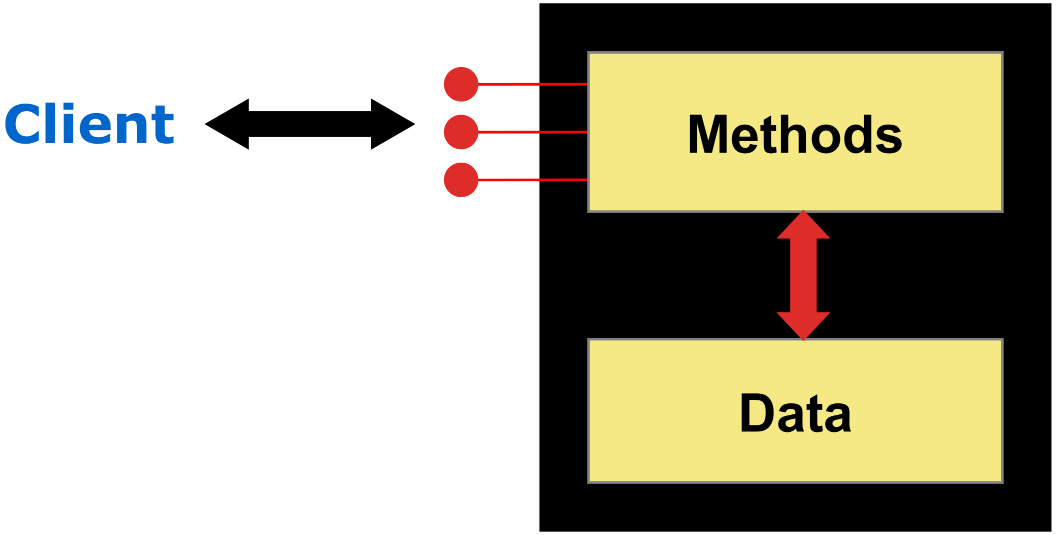
- Lecture 4 Review
- Inheritance
- Object as Superclass
- Casting Objects
- Abstract Methods and Classes

Lecture 4 Review

Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client

- The client invokes the interface methods of the object, which manages the instance data



Visibility Modifiers for Encapsulation

- `public`
- `protected`
- `private`
  - can be referenced only within that class
- `public` variables violate encapsulation
  - clients modify the values directly!!
  - Instance variables should not be declared public
- *Service methods* are `public` (for clients)
- *Support methods* are not `public` (for service methods)

Accessors and Mutators

- Because instance data is private, a class usually provides services to access and modify data values

- *Accessor method* returns the current value of a variable (`getX,` where `X` is the name of the value)

- *Mutator method* changes the value of a variable (`setX`)

Avoiding shadowing w/ `this`

```
public class Point {
    private int x;
    private int y;

    ...

    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

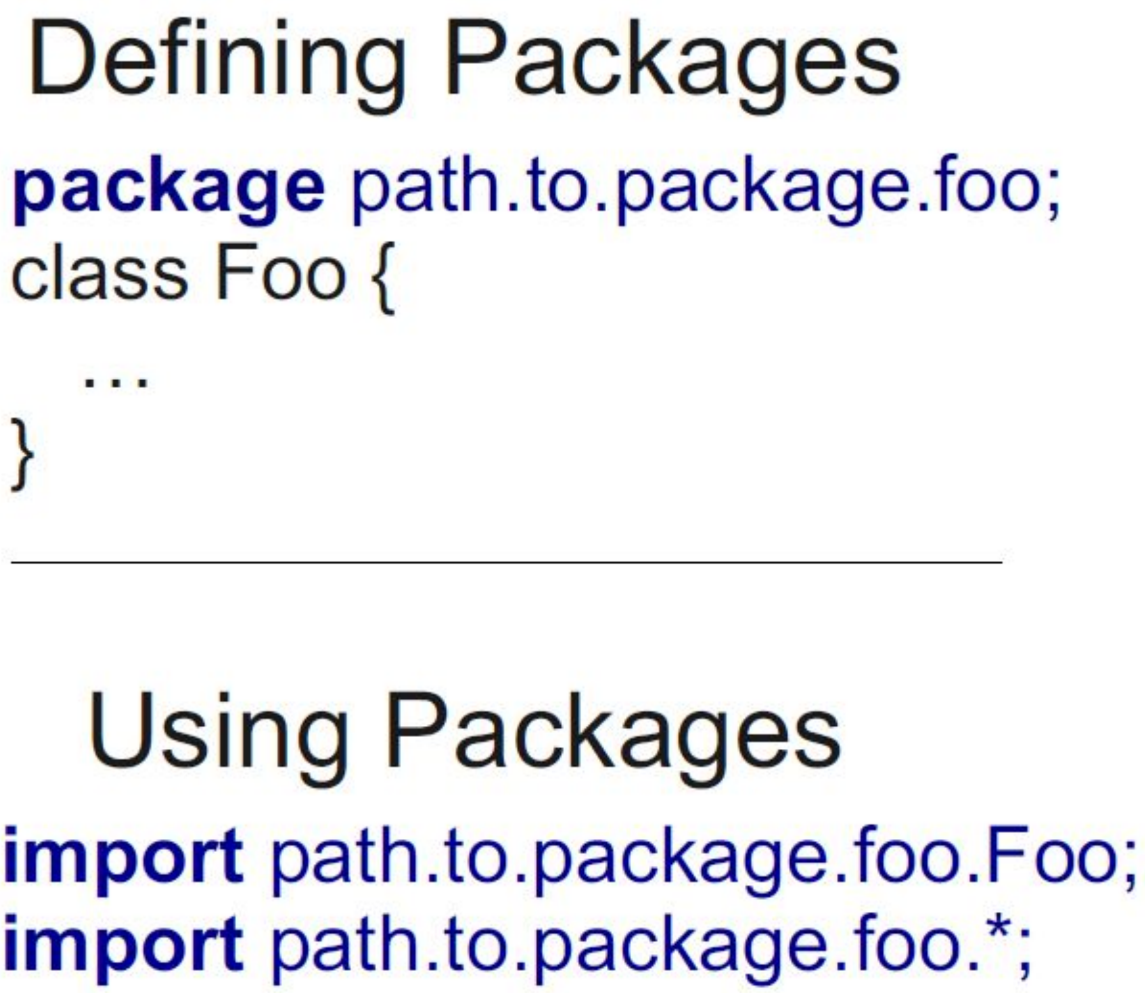
- Inside the `setLocation` method,
  - When `this.x` is seen, the *field* `x` is used.
  - When `x` is seen, the *parameter* `x` is used.

Constructors and `this`

- One constructor can call another using `this`:

```
public class Point {
    private int x;
    private int y;

    public Point() {
        this(0, 0);  // calls the (x, y) constructor
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```

Packages



Defining Packages

```
package path.to.package.foo;
class Foo {
    ...
}
```

Using Packages

```
import path.to.package.foo.Foo;
import path.to.package.foo.*;
```

Why Packages?

- Group similar functionality
  - org.boston.libraries.Library
  - org.boston.libraries.Book
- Seperate similar names
  - shopping.List
  - packaging.List

Special Packages

- All classes see classes in the same package (No need to import)

- All classes see classes in java.lang
  - Example: java.lang.String; java.lang.System

Java API

- Java includes lots of packages/classes
- Reuse classes to avoid extra work
- http://docs.oracle.com/javase/8/docs/api/

Inheritance

Review: Classes

- User-defined data types
  - Defined using the “class” keyword
  - Each class has associated
    - Variables (any object type)
    - Methods that operate on the data (variables)
- New instances of the class are declared using the “new” keyword
- “Static” variables/methods have only one copy, regardless of how many instances are created

Example: Shared Functionality

```
public class Student {
  String name;
  char gender;
  Date birthday;
  ArrayList<Grade> grades;

  double getGPA() {
    …
  }

  int getAge(Date today) {
    …
  }
}
```

```
public class Professor {
  String name;
  char gender;
  Date birthday;
  ArrayList<Paper> papers;

  int getCiteCount() {
    …
  }

  int getAge(Date today) {
    …
  }
}
```

```java
public class Person {
  String name;
  char gender;
  Date birthday;

  int getAge(Date today) {
    …
  }
}
```

```java
public class Student
      extends Person {

  ArrayList<Grade> grades;

  double getGPA() {
    …
  }
}
```

```java
public class Professor
      extends Person {

  ArrayList<Paper> papers;

  int getCiteCount() {
    …
  }
}
```

Inheritance

- “is-a” relationship
- Single inheritance:
  - Subclass is derived from one existing class (superclass)
- Multiple inheritance:
  - Subclass is derived from more than one superclass
  - Not supported by Java
  - A class can only extend the definition of one class

Inheritance (continued)

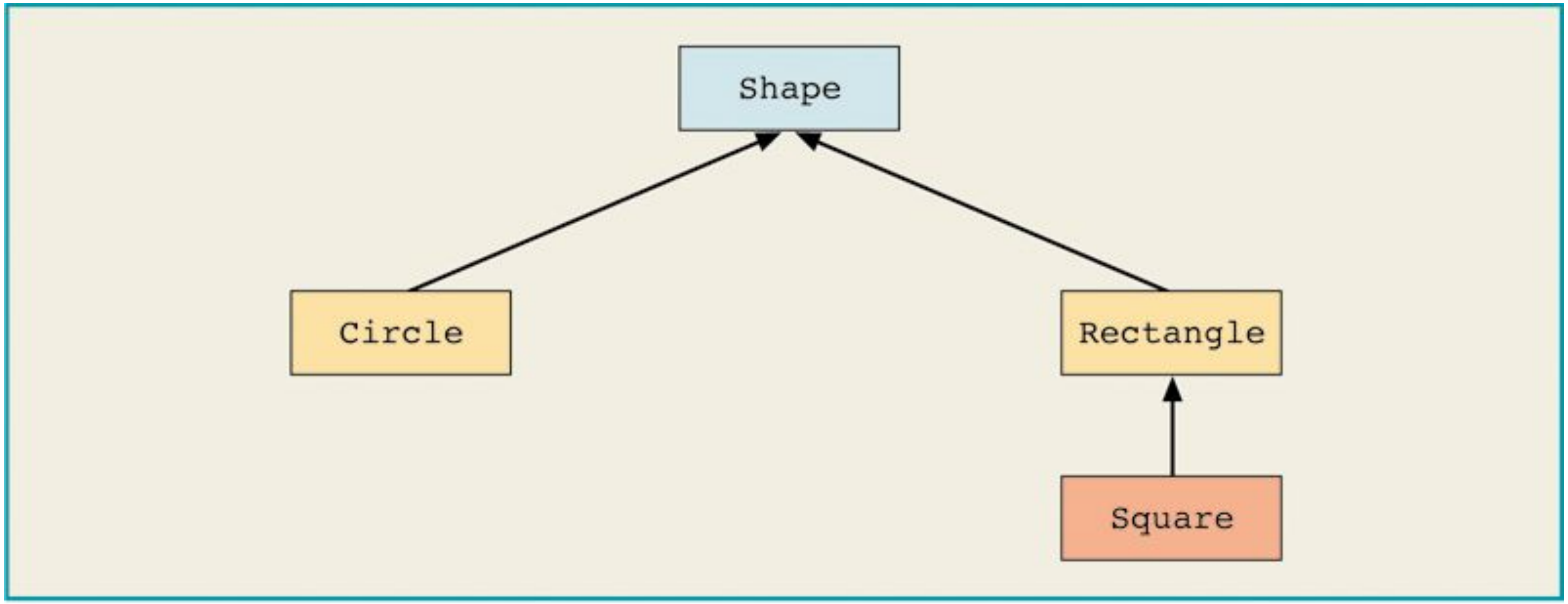


Figure 11-1 Inheritance hierarchy

```
public class ClassName extends ExistingClassName
{
    memberList
}
```

Inheritance: `class Circle` Derived from `class Shape`

```
public class Circle extends Shape
{
          .
          .
          .
}
```

Inheritance

- Allow us to specify *relationships between types*

- Why is this useful in programming?
  - Allows for code reuse
  - Polymorphism

Code Reuse

- General functionality can be written once and applied to **any** subclass
- Subclasses can specialize by adding members and methods, or overriding functions

Inheritance: Adding Functionality

- Subclasses have ***all*** of the data members and methods of the superclass
- Subclasses can add to the superclass
  - Additional data members
  - Additional methods
- Subclasses are more specific and have more functionality
- Superclasses capture generic functionality common across many types of objects

```java
public class Person {
  String name;
  char gender;
  Date birthday;

  int getAge(Date today) {
    …
  }
}
```

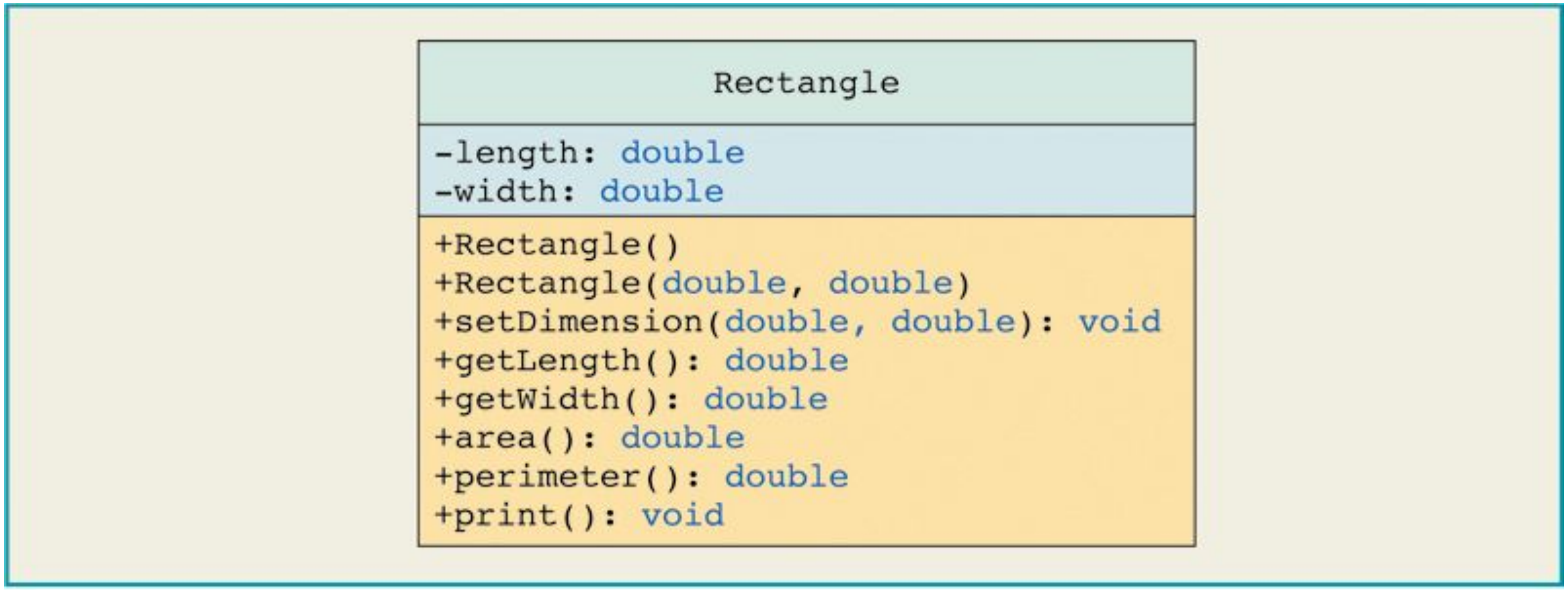
```java
public class Student
      extends Person {

  ArrayList<Grade> grades;

  double getGPA() {
    …
  }
}
```

```java
public class Professor
      extends Person {

  ArrayList<Paper> papers;

  int getCiteCount() {
    …
  }
}
```

Brainstorming

- What are some other examples of possible inheritance hierarchies?
  - Person -> student, faculty…
  - Shape -> circle, triangle, rectangle…
  - Other examples???

UML Diagram: `Rectangle`



```
Rectangle
-length: double
-width: double
+Rectangle()
+Rectangle(double, double)
+setDimension(double, double): void
+getLength(): double
+getWidth(): double
+area(): double
+perimeter(): double
+print(): void
```

Figure 11-2 UML class diagram of the `class Rectangle`

What if we want to implement a 3d box object?

Objects `myRectangle` and `myBox`

```
Rectangle myRectangle = new Rectangle(5, 3);
Box myBox = new Box(6, 5, 4);
```

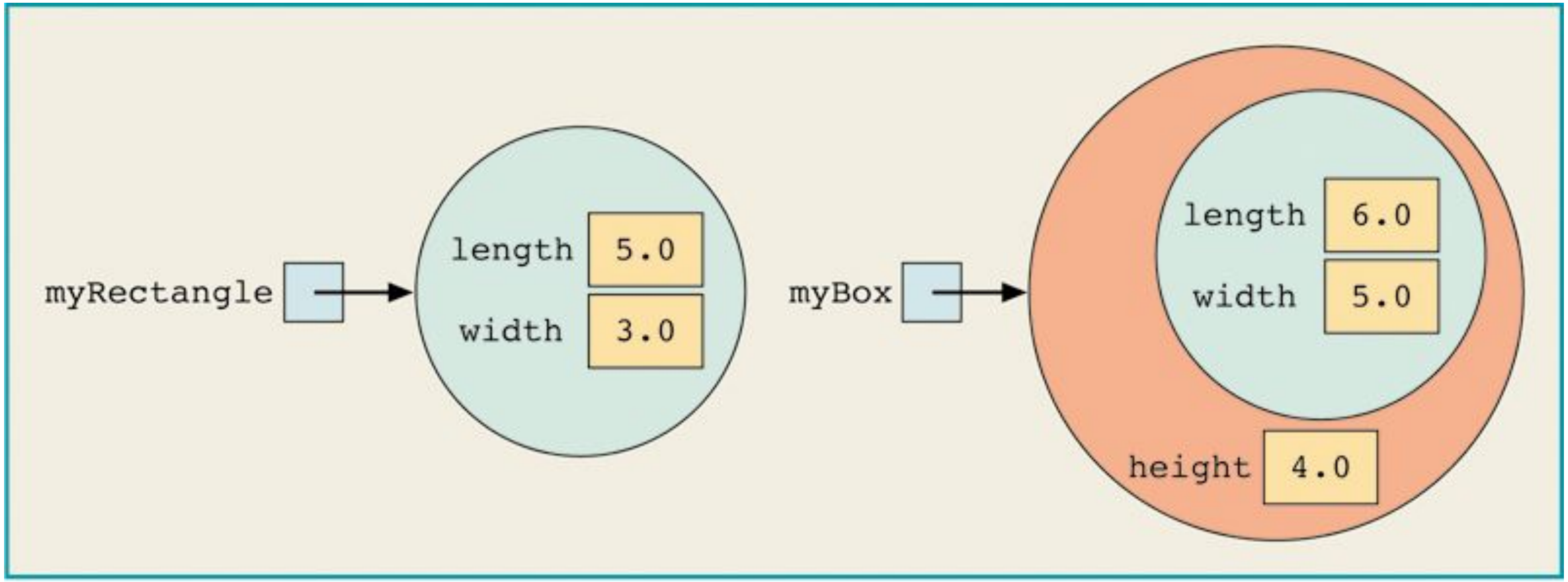
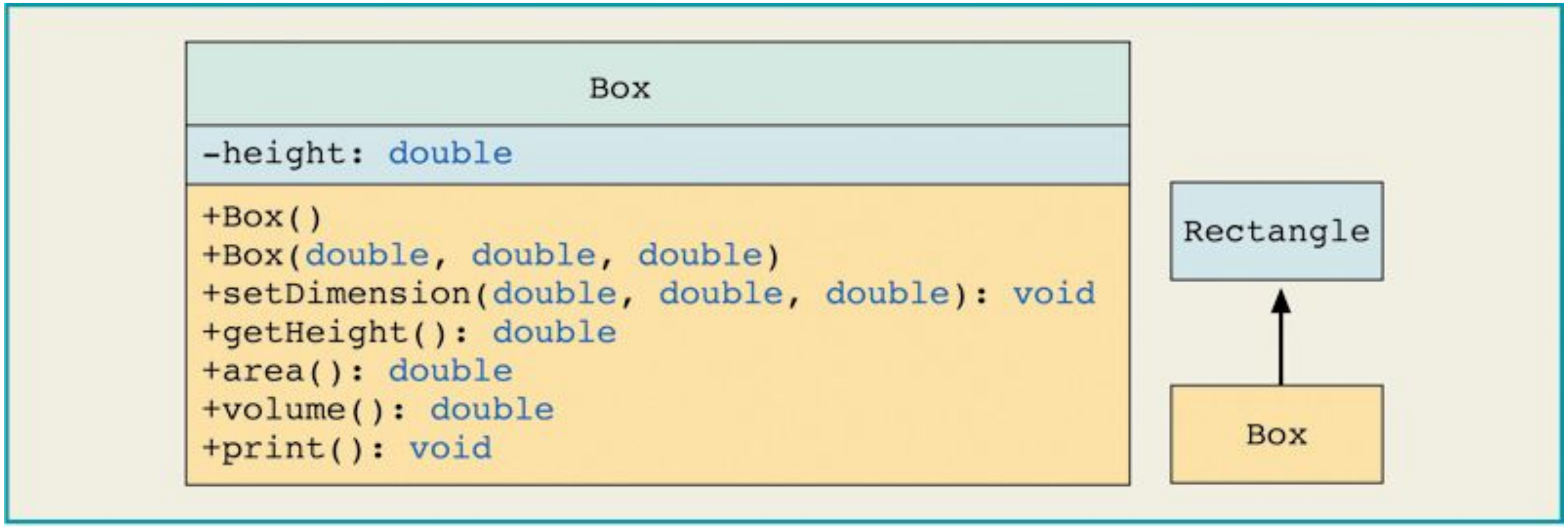


Figure 11-4 Objects `myRectangle` and `myBox`

UML Class Diagram: class Box



```
Box
-height: double
+Box()
+Box(double, double, double)
+setDimension(double, double, double): void
+getHeight(): double
+area(): double
+volume(): double
+print(): void
```

Figure 11-3 UML class diagram of the **class** Box and the inheritance hierarchy

Both a Rectangle and a Box have a surface area, but they are computed differently

Overriding Methods

- A subclass can override (redefine) the methods of the superclass
  - Objects of the subclass type will use the new method
  - Objects of the superclass type will use the original

class Rectangle

```
public double area()
{
    return  getLength() * getWidth();
}
```

class Box

```
public double area()
{
    return  2 * (getLength() * getWidth()
                 + getLength() * height
                 + getWidth() * height);
}
```

final Methods

- Can declare a method of a class final using the keyword final

```
public final void doSomeThing()
{
        //...
}
```

- If a method of a class is declared final, it cannot be overridden with a new definition in a derived class

Modifiers

- A subclass does not inherit/access the **private** members of its parent class.

Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Modifiers

- The access specifier for an overriding method can allow more, but not less, access than the overridden method.
  - a protected instance method in the superclass can be made public, but not private, in the subclass.

- You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.

Hiding Fields

- Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different.
- Hiding fields is not recommended as it makes code difficult to read.

Calling methods of the superclass

- To write a method's definition of a subclass, specify a call to the public method of the superclass
  - If subclass overrides public method of superclass, specify call to public method of superclass:

    ```
    super.MethodName(parameter list)
    ```

  - If subclass does not override public method of superclass, specify call to public method of superclass:

    ```
    MethodName(parameter list)
    ```

class Box

```
public void setDimension(double l, double w, double h)
{
    super.setDimension(l, w);
    if (h >= 0)
        height = h;
    else
        height = 0;
}}
```

Box overloads the method setDimension
(Different parameters)

Method *Overloading*

- **Method overloading:** *multiple* methods ...
  - With the *same name*
  - But *different signatures*
  - In the *same class*
- Constructors are often overloaded
- Example:
  - **`MyClass (int inputA, int inputB)`**
  - **`MyClass (float inputA, float inputB)`**

Overloading Example From Java Library

`ArrayList` has two **`remove`** methods:

`remove (int position)`

- Removes object that is at a specified *place* in the list

`remove (Object obj)`

- Removes a *specified object* from the list

It also has two **`add`** methods:

`add (Element e)`

- Adds new object to the *end* of the list

`add (int index, Element e)`

- Adds new object at a *specified place* in the list

Defining Constructors of the Subclass

- Call to constructor of superclass:
  - Must be first statement
  - Specified by super parameter list

```
public Box()
{
    super();
    height = 0;
}

public Box(double l, double w, double h)
{
    super(l, w);
    height = h;
}
```

38

Object as a Superclass

- **`Object`** is the root of the class hierarchy
  - Every *class* has **`Object`** as a superclass
- All classes inherit the methods of Object
  - But may override them

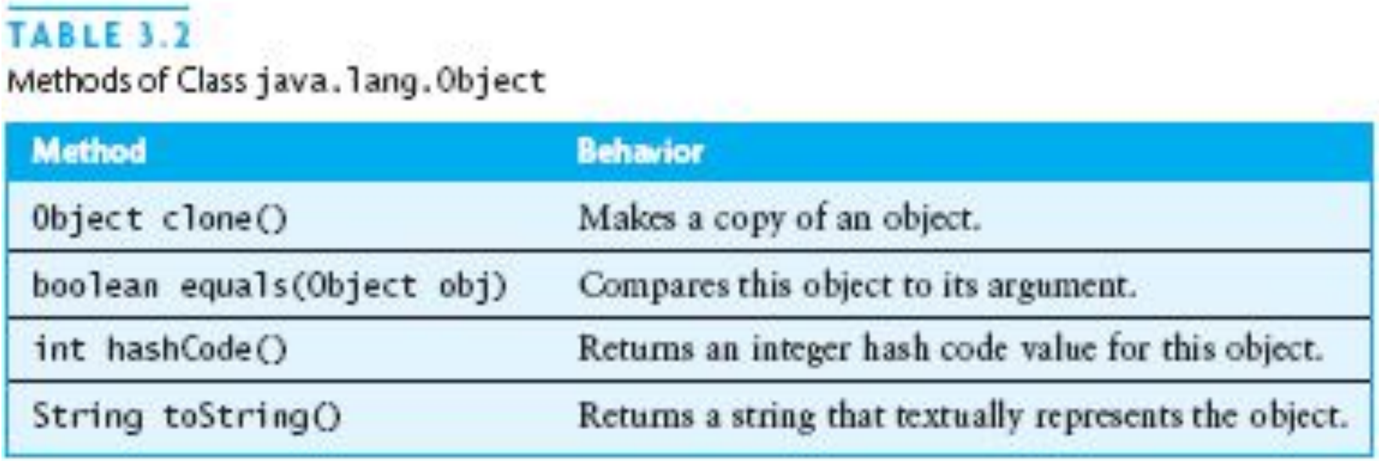


TABLE 3.2

Methods of Class `java.lang.Object`

Method	Behavior
`Object clone()`	Makes a copy of an object.
`boolean equals(Object obj)`	Compares this object to its argument.
`int hashCode()`	Returns an integer hash code value for this object.
`String toString()`	Returns a string that textually represents the object.

The **`toString() Method`**

- The Object's toString() method returns a String representation of the object, which is very useful for debugging.

- You should always override **`toString`** method if you want to print object state

- If you do *not* override it:
  - **`Object.toString`** will return a **`String`**
  - Just not the **`String`** you want!

    Example: **`ArrayBasedPD@ef08879`**

    ... The name of the class, `@`, instance’s hash code

The **`equals()`** Method

- Compares two objects for equality and returns true if they are equal.

- The equals() method provided by Object tests whether the object references are equal—that is, if the objects compared are the exact same object.

- To test whether two objects are equal in the sense of containing the same information, you must override the equals() method.

The **`hashCode()`** Method

- The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal.

- By definition, if two objects are equal, their hash code must also be equal.

- If you override the equals() method, you must also override the hashCode() method as well.

The **`getClass()`** Method

- The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), etc..

Operations Determined by Type of Reference Variable

- Variable can refer to object whose type is a *subclass* of the variable’s declared type
- Type of the *variable* determines what operations are legal
- Java is *strongly typed*
  - Compiler always verifies that variable’s type includes the class of every expression assigned to the variable

```
Object obj= new Box(5,5,);
obj.area();                        //  compile-time error.
```

Casting Objects

- *Casting* obtains a reference of different, but *matching,* type
- Casting *does not change* the object!
  - It creates an anonymous reference to the object

```
Box box= (Box)obj;
```

- *Downcast:*
  - Cast *superclass* type to *subclass* type
  - Checks *at run time* to make sure it's ok
  - If not ok, throws **`ClassCastException`**

Casting Objects

- Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance

```
Box box= (Box)obj; //compile-time error
```

- would get a compile-time error because obj is not known to the compiler to be Box

- However, we can tell the compiler that we promise to assign a MountainBike to obj by explicit casting:

instanceof operator

- **instanceof** can guard against **ClassCastException**

```
Object obj = ...;
if (obj instanceof Box) {
  Box box = (Box)obj;
  int area= box.area();
  ...;
} else {
  ...
}
```

Abstract Methods and Classes

- An abstract class is a class that is declared abstract
  - it may or may not include abstract methods.

- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

Abstract Methods and Classes

```
public abstract class Shape{

   // declare fields

   // declare nonabstract methods

   abstract void calculateArea();
   abstract void calculatePerimeter();
}
```

Abstract Methods and Classes

- When an abstract class is subclassed,
  - the subclass usually provides implementations for all of the abstract methods in its parent class.
  - if it does not, then the subclass must also be declared abstract.

Abstract Methods and Classes

```
class Circle extends Shape{
    void calculateArea() {
        ...
    }
    void calculatePerimeter() {
        ...
    }
}
```

Summary for today

- Inheritance
- Object as Superclass
- Casting Objects
- Abstract Methods and Classes

What You Can Do in a Subclass

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.

What You Can Do in a Subclass

- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
- You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.

What You Can Do in a Subclass

- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super

Other Issues

- Except for the Object class, a class has exactly one direct superclass.

- The Object class is the top of the class hierarchy. All classes are descendants from this class and inherit methods from it. Useful methods inherited from Object include
  - toString(), equals(), clone(), and getClass().

Other Issues

- You can prevent a class from being subclassed by using the final keyword in the class's declaration.
- Similarly, you can prevent a method from being overridden by subclasses by declaring it as a final method.
- An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods

References

- http://math.hws.edu/javanotes/
- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/
- https://docs.oracle.com/javase/tutorial/java
- http://www.cs.utep.edu/vladik/cs2401.10a/Ch_11_Inheritance_Polymorphism.ppt
- https://people.cs.umass.edu/~moss/187/lectures/lecture-d-inheritance.ppt