

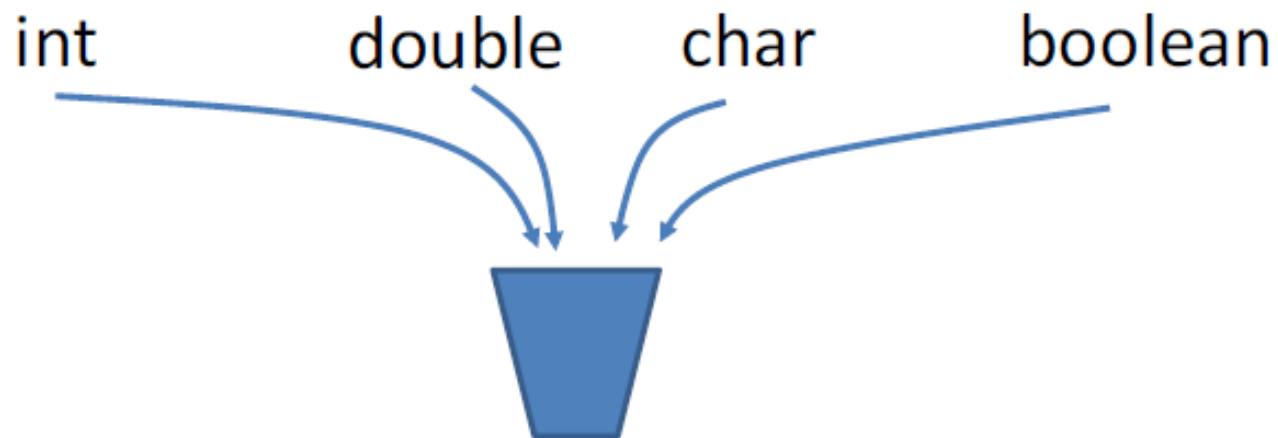
References vs Values

Primitives vs References

- **Primitive** types are basic java types
 - int, long, double, boolean, char, short, byte, float
 - The actual **values** are stored in the variable
- **Reference** types are arrays and objects
 - String, int[], Baby, ...|

How java stores **primitives**

- Variables are like fixed size cups
- Primitives are small enough that they just fit into the cup



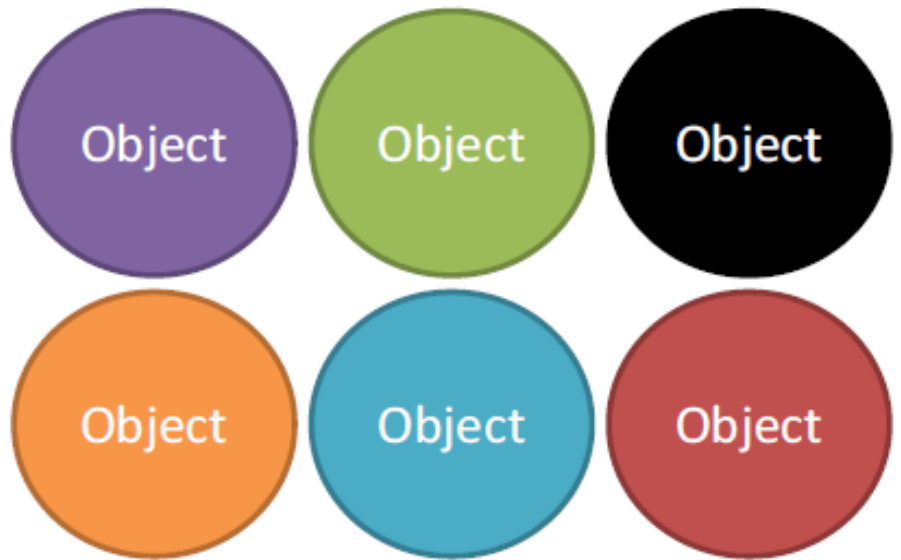
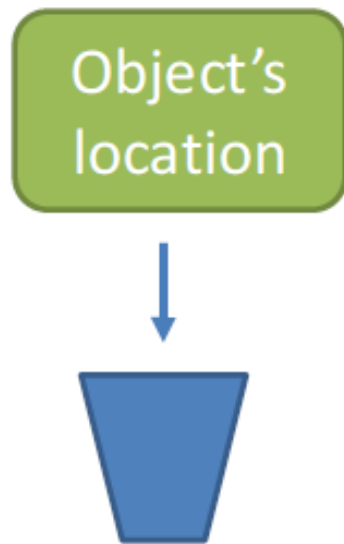
How java stores **objects**

- Objects are too big to fit in a variable
 - Stored somewhere else
 - Variable stores a number that locates the object



How java stores **objects**

- Objects are too big to fit in a variable
 - Stored somewhere else
 - Variable stores a number that locates the object



References

- The object's location is called a reference
- == operator compares the references

```
Point point1 = new Point(10,10);  
Point point2 = new Point(10, 10);
```

Does point1 ==point2 ?

References

- The object's location is called a reference
- == operator compares the references

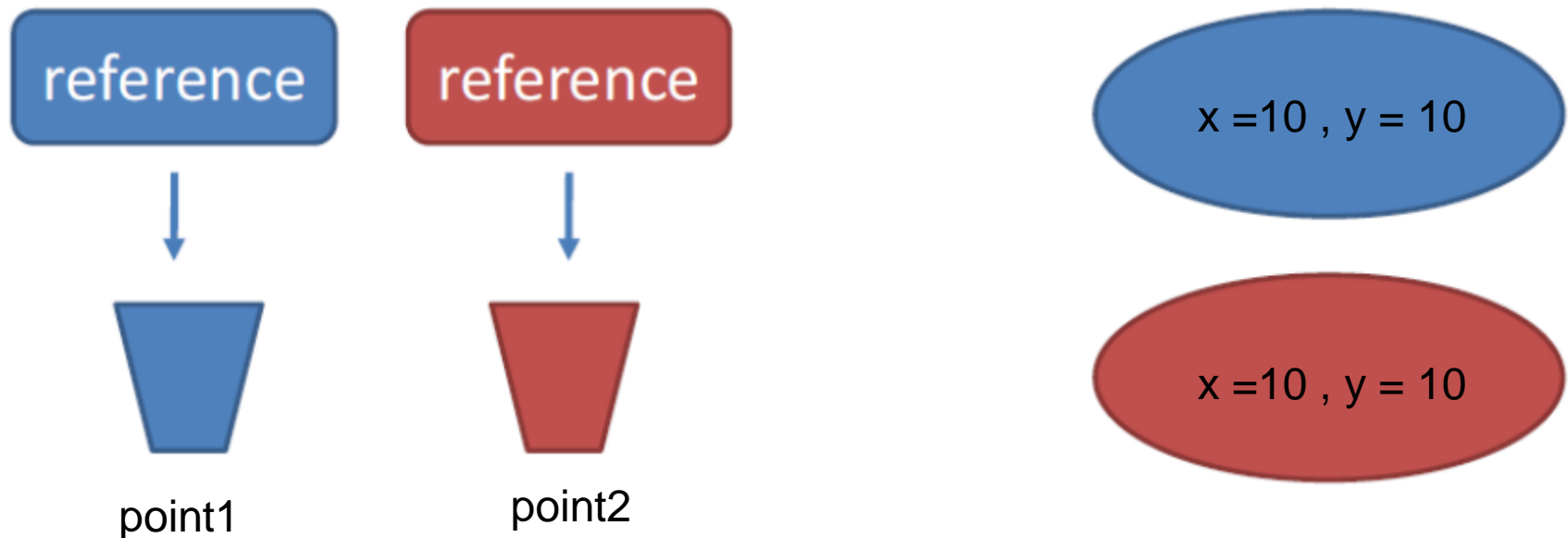
```
Point point1 = new Point(10,10);  
Point point2 = new Point(10, 10);
```

Does point1 ==point2 ?

NO

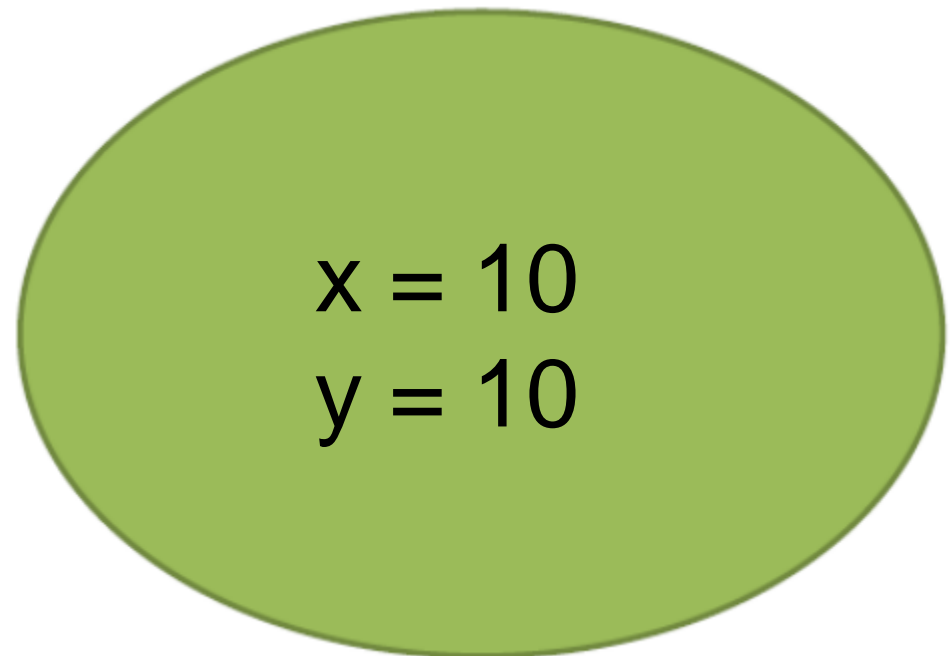
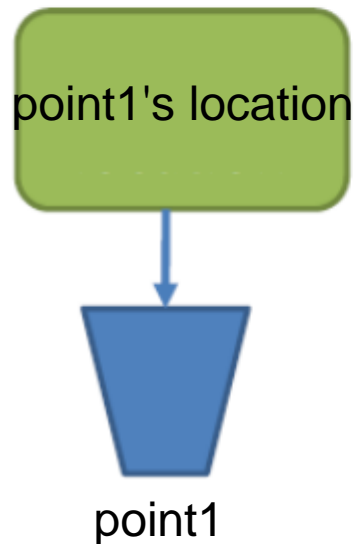
References

```
Point point1 = new Point(10,10);  
Point point2 = new Point(10, 10);
```



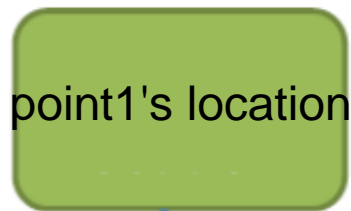
References

- `Point point1 = new Point(10,10);`

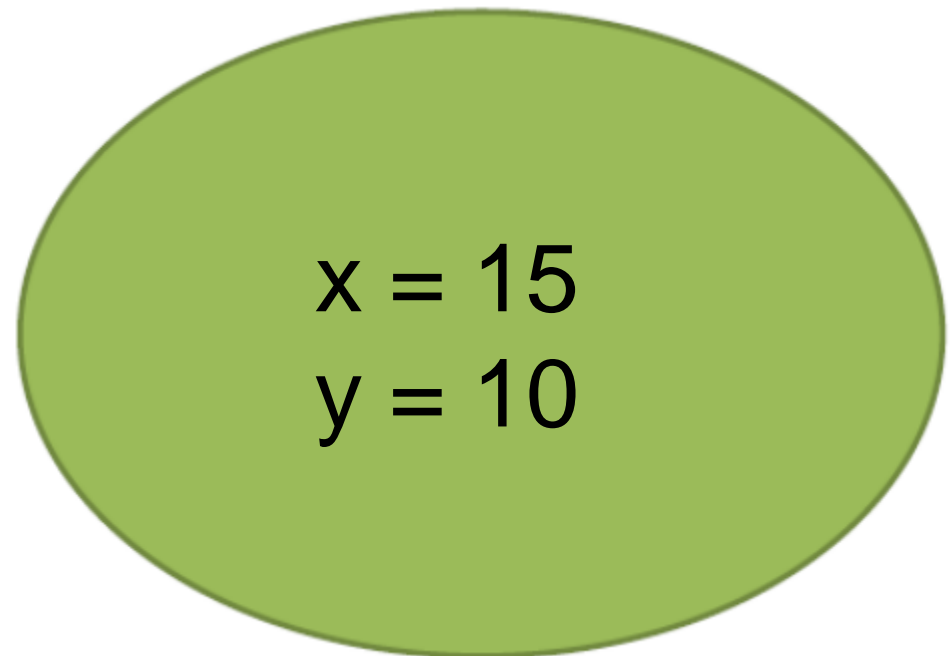


References

- `Point point1 = new Point(10,10);`
- `point1.x = 15;`

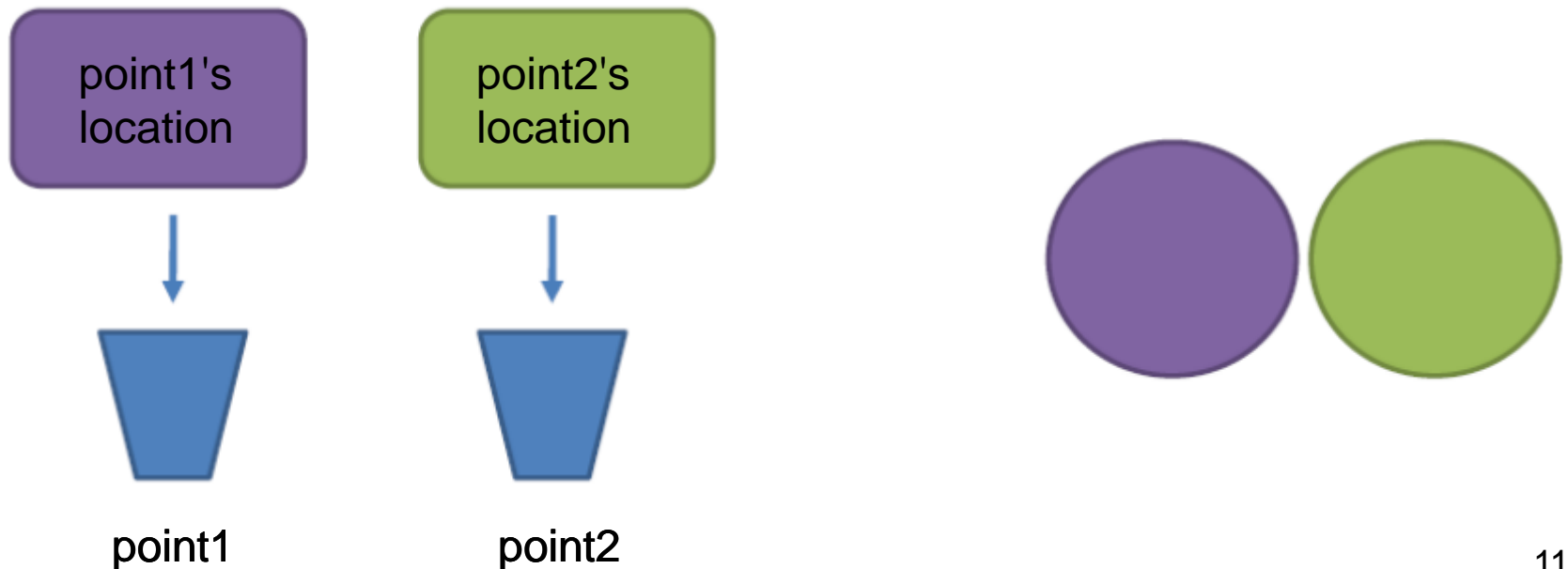


point1



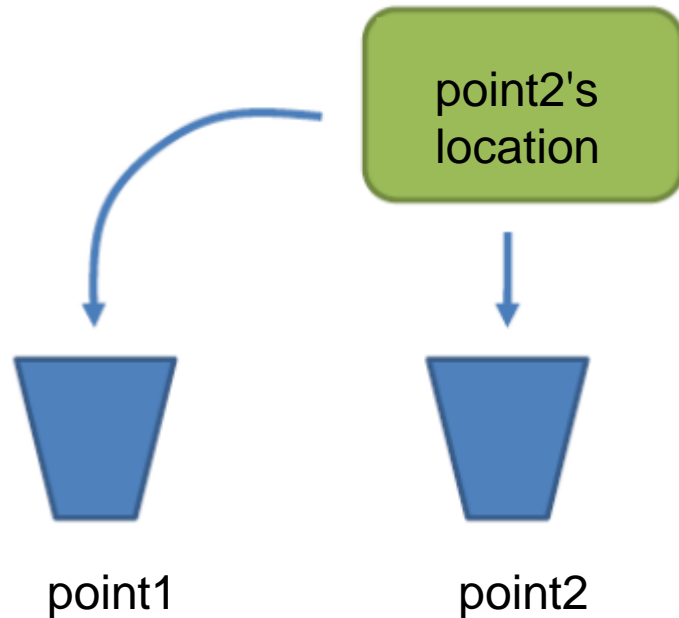
References

```
Point point1 = new Point(10,10);  
Point point2 = new Point(10, 10);
```



References

```
point1 = point2;
```



Constructors (revisited)

- All classes have constructors, whether you define one or not
- Java automatically provides a default constructor, if constructor is not declared
 - non-initialized member variables have their default values
- Once you define your own constructor, the default constructor is no longer used.

Constructors (revisited)

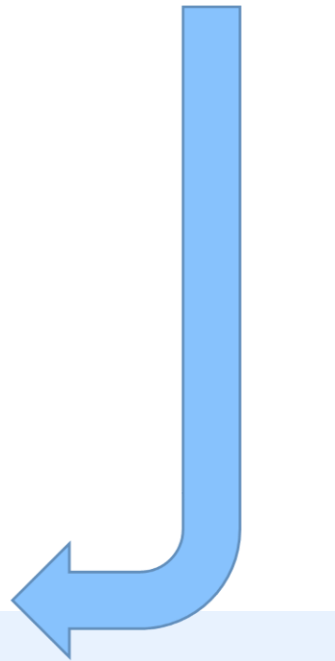
- Is there a constructor here

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

Constructors (revisited)

- Yes! The default constructor is provided by Java.

```
public class Point {  
  
    int x;  
    int y;  
  
    public Point() {  
          
    }  
  
}
```



Constructors (revisited)

- Once you define your own constructor, the default constructor is no longer available.

```
public class Point {  
  
    int x;  
    int y;  
  
    public Point(int x, int y) {  
  
    }  
  
}
```


Constructors (revisited)

- non-initialized member variables

```
public class Circle {  
    int radius;        //default value is 0  
    Point center;      //default value is null  
}
```

Constructors (revisited)

```
// A simple constructor.
```

```
class MyClass {  
    int x;
```

```
    MyClass() { ← This is the constructor for MyClass.  
        x = 10;  
    }
```

```
}
```

```
class ConsDemo {
```

```
    public static void main(String args[]) {  
        MyClass t1 = new MyClass();  
        MyClass t2 = new MyClass();
```

```
        System.out.println(t1.x + " " + t2.x);
```

```
    }
```

```
}
```

Parametrized Constructors

```
// A parameterized constructor.
```

```
class MyClass {  
    int x;
```

```
    MyClass(int i) { ←———— This constructor has a parameter.  
        x = i;
```

```
    }  
}
```

```
class ParmConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass(10);  
        MyClass t2 = new MyClass(88);  
  
        System.out.println(t1.x + " " + t2.x);  
    }  
}
```

static fields and methods

static field

- Applies to fields and methods
- Means the field/method
 - Is defined for the class declaration,
 - Is not unique for each instance

static field

- Keep track of the number of points

```
public class Point {  
  
    int xCoordinate;  
    int yCoordinate;  
  
    static int count = 0;  
  
    public Point(int x, int y){  
        xCoordinate = x;  
        yCoordinate = y;  
    }  
  
    public void move(int xDistance, int yDistance){  
        xCoordinate += xDistance;  
        yCoordinate += yDistance;  
    }  
}
```

static field

- Keep track of the number of points

```
public class Point {  
  
    int xCoordinate;  
    int yCoordinate;  
  
    static int count;  
  
    public Point(int x, int y){  
        xCoordinate = x;  
        yCoordinate = y;  
        count ++;  
    }  
  
    public void move(int xDistance, int yDistance){  
        xCoordinate += xDistance;  
        yCoordinate += yDistance;  
    }  
}
```

static field

```
public static void main(String[] args) {  
  
    System.out.println(Point.count);  
  
    Point point1 = new Point(10,10);  
    Point point2 = new Point(15, 22);  
  
    System.out.println(Point.count);  
  
}
```


static field

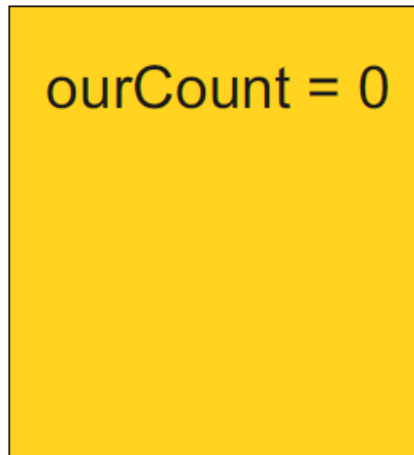
```
public class Counter {  
    int myCount = 0;  
    static int ourCount = 0;  
    void increment() {  
        myCount++;  
        ourCount++;  
    }  
    public static void main(String[] args) {  
        Counter counter1 = new Counter();  
        Counter counter2 = new Counter();  
        counter1.increment();  
        counter1.increment();  
        counter2.increment();  
        System.out.println("Counter 1: " +  
counter1.myCount + " " + counter1.ourCount);  
        System.out.println("Counter 2: " +  
counter2.myCount + " " + counter2.ourCount);  
    }  
}
```

static field

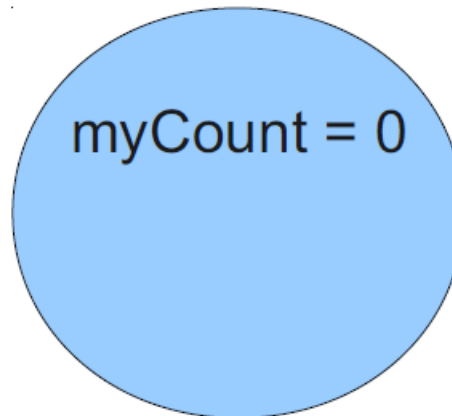
```
public class Counter {  
    int myCount = 0;  
    static int ourCount = 0; Fields  
    void increment() {  
        myCount++;  
        ourCount++; Method  
    }  
    public static void main(String[] args) {  
        Counter counter1 = new Counter();  
        Counter counter2 = new Counter();  
        counter1.increment();  
        counter1.increment();  
        counter2.increment();  
        System.out.println("Counter 1: " +  
            counter1.myCount + " " + counter1.ourCount);  
        System.out.println("Counter 2: " +  
            counter2.myCount + " " + counter2.ourCount);  
    }  
}
```

static field

Class Counter



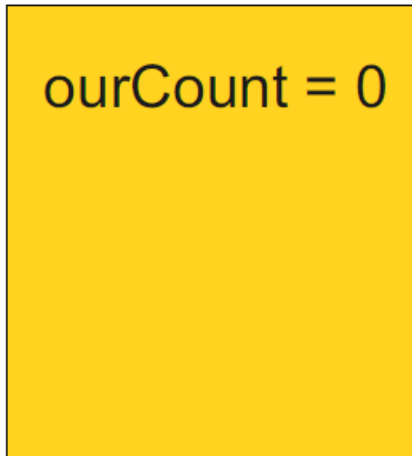
Object counter1



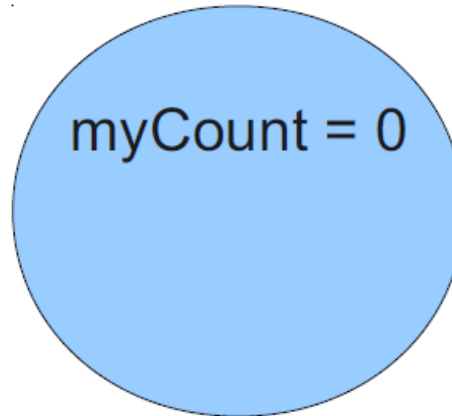
```
Counter counter1 = new Counter();
```

static field

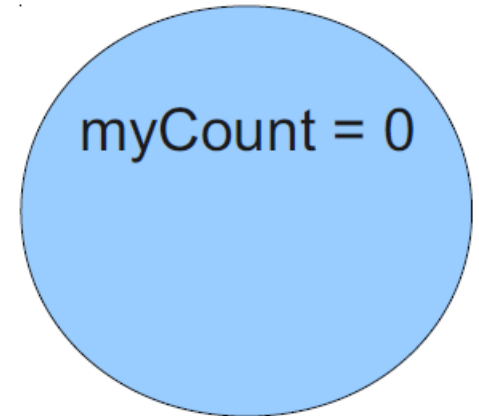
Class Counter



Object counter1



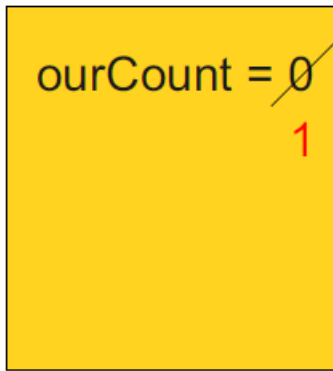
Object counter2



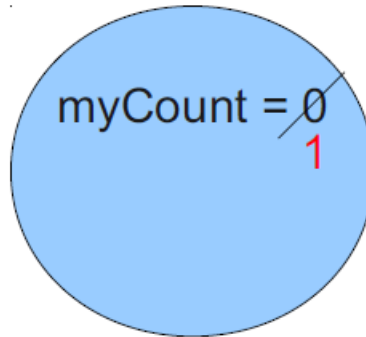
```
Counter counter1 = new Counter();  
Counter counter2 = new Counter();
```

static field

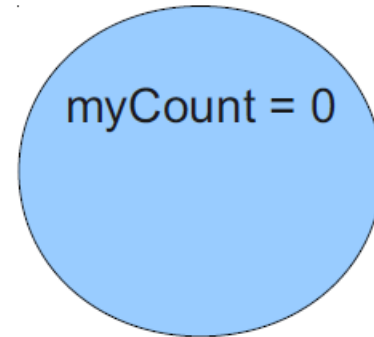
Class Counter



Object counter1



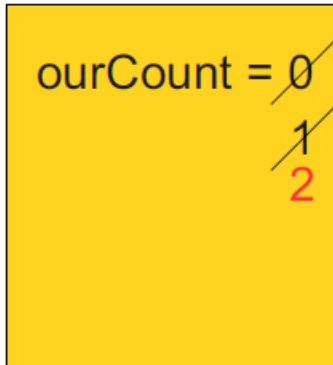
Object counter2



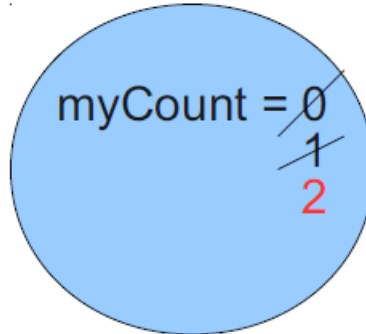
```
Counter counter1 = new Counter();  
Counter counter2 = new Counter();  
counter1.increment();
```

static field

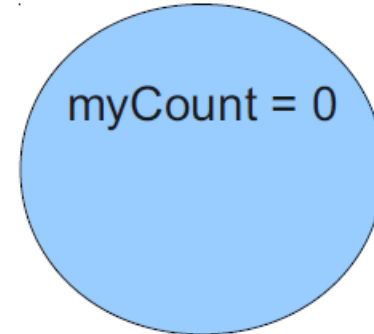
Class Counter



Object counter1



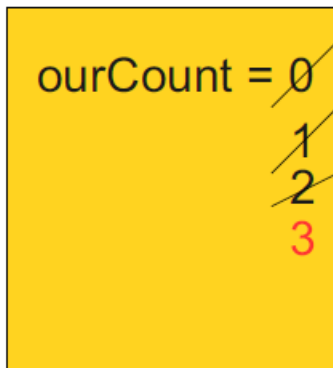
Object counter2



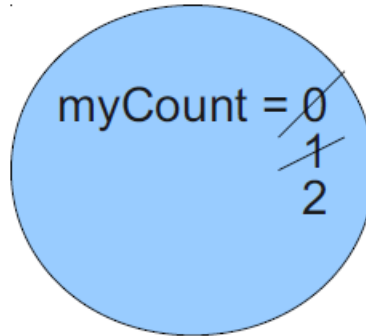
```
Counter counter1 = new Counter();  
Counter counter2 = new Counter();  
counter1.increment();  
counter1.increment();
```

static field

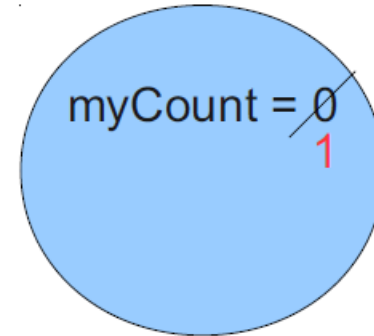
Class Counter



Object counter1



Object counter2



```
Counter counter1 = new Counter();  
Counter counter2 = new Counter();  
counter1.increment();  
counter1.increment();  
counter2.increment();
```

Access control

Access Control

```
public class CreditCard {  
    String cardNumber;  
    double expenses;  
    void charge(double amount) {  
        expenses = expenses + amount;  
    }  
    String getCardNumber(String password) {  
        if (password.equals("SECRET!3*!")) {  
            return cardNumber;  
        }  
        return "jerkface";  
    }  
}
```

Malicious

```
public class Malicious {  
    public static void main(String[] args) {  
        maliciousMethod(new CreditCard());  
    }  
    static void maliciousMethod(CreditCard card)  
    {  
        card.expenses = 0;  
        System.out.println(card.cardNumber);  
    }  
}
```

Public vs. Private

- Public: others can use this
- Private: only the class can use this

public/private applies to any
field or **method**

Access Control

```
public class CreditCard {  
    String cardNumber;  
    double expenses;  
    void charge(double amount) {  
        expenses = expenses + amount;  
    }  
    String getCardNumber(String password) {  
        if (password.equals("SECRET!3*!")) {  
            return cardNumber;  
        }  
        return "jerkface";  
    }  
}
```

Access Control DONE

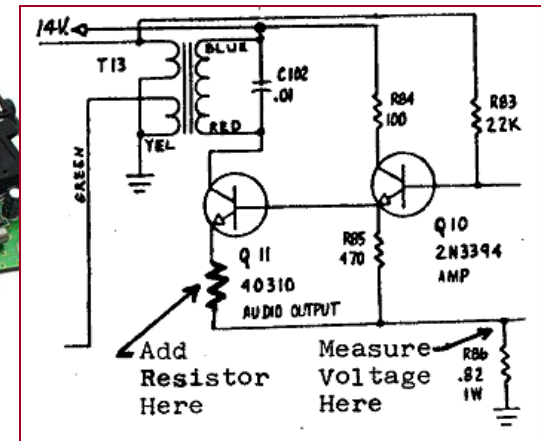
```
public class CreditCard {  
    private String cardNumber;  
    private double expenses;  
    public void charge(double amount) {  
        expenses = expenses + amount;  
    }  
    public String getCardNumber(String password)  
    {  
        if (password.equals("SECRET!3*!")) {  
            return cardNumber;  
        }  
        return "jerkface";  
    }  
}
```

Why Access Control

- Protect private information
- Clarify how others should use your class
- Keep implementation separate from interface

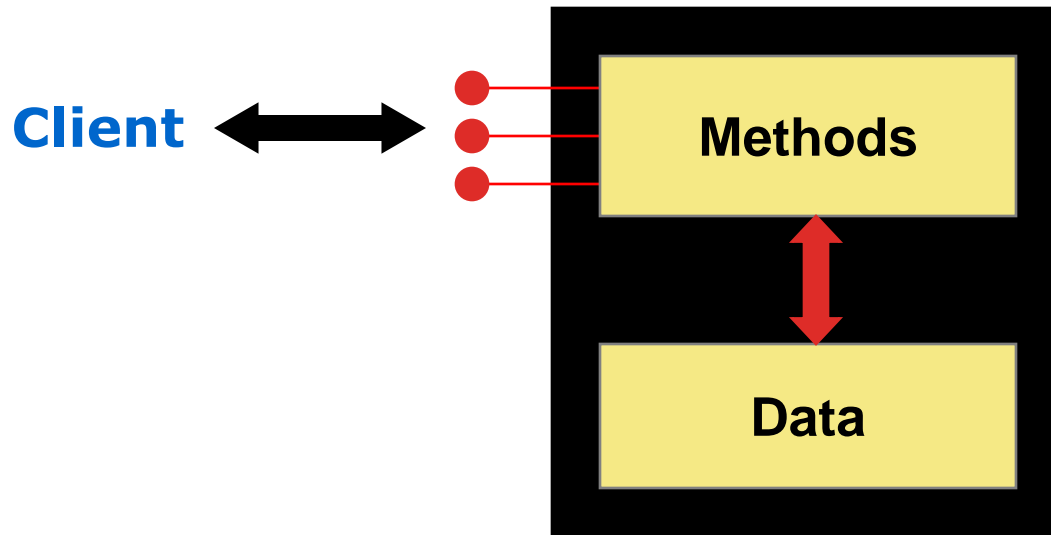
Encapsulation

- **encapsulation:** Hiding implementation details of an object from its clients.
 - Encapsulation provides *abstraction*.
 - separates external view (behavior) from internal view (state)
 - Encapsulation protects the integrity of an object's data.



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



Private fields

- A field can be declared *private*.
 - No code outside the class can access or change it.

```
private type name;
```

- Examples:

```
private int id;  
private String name;
```

- Client code sees an error when accessing private fields:

```
PointMain.java:11: x has private access in Point  
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");  
                        ^
```

Accessing private state

- We can provide methods to get and/or set a field's value:

```
// A "read-only" access to the x  
field ("accessor")
```

```
public int getX() {  
    return x;  
}
```

```
// Allows clients to change the x  
field ("mutator")
```

```
public void setX(int newX) {  
    x = newX;  
}
```

Point class

// A Point object represents an (x, y) location.

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    public double distanceFromOrigin() {  
        return Math.sqrt(x * x + y * y);  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setLocation(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
  
    public void translate(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
}
```

Client code

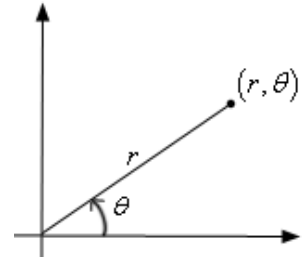
```
public class PointMain {  
    public static void main(String[] args) {  
        // create two Point objects  
        Point p1 = new Point(5, 2);  
        Point p2 = new Point(4, 3);  
  
        // print each point  
        System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");  
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");  
  
        // move p2 and then print it again  
        p2.translate(2, 4);  
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");  
    }  
}
```

OUTPUT:

```
p1 is (5, 2)  
p2 is (4, 3)  
p2 is (6, 7)
```

Benefits of encapsulation

- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.
 - A bank app forbids a client to change an `Account`'s balance.
- Allows you to change the class implementation.
 - `Point` could be rewritten to use polar coordinates (radius r , angle θ), but with the same methods.
- Allows you to constrain objects' state.
 - Example: Only allow `Points` with non-negative coordinates.



Class Scope

Scope Review

```
public class ScopeReview {  
    void scopeMethod(int var1) {  
        String var2;  
        if (var1 > 0) {  
            var2 = "above 0";  
        } else {  
            var2 = "less than or equal to 0";  
        }  
        System.out.println(var2);  
    }  
}
```

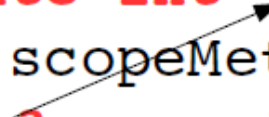
The diagram illustrates variable scope resolution in the provided Java code. It features two arrows: one originates from the 'var1' parameter in the method signature 'void scopeMethod(int var1)' and points to its usage 'var1 > 0' within the 'if' statement; the other originates from the 'var2' local variable declaration 'String var2;' and points to its usage 'var2 = \"above 0\";' within the 'if' block. This visualizes how the local 'var2' is resolved when used within its own scope, while 'var1' is resolved to the parameter in the enclosing method scope.

Scope Review

```
public class ScopeReview {  
    private int var3;  
    void scopeMethod(int var1) {  
        var3 = var1;  
        String var2;  
        if (var1 > 0) {  
            var2 = "above 0";  
        } else {  
            var2 = "less than or equal to 0";  
        }  
        System.out.println(var2);  
    }  
}
```


Class Scope

```
public class ScopeReview {  
    private int var3;  
    void scopeMethod(int var1) {  
        var3 = var1;  
        String var2;  
        if (var1 > 0) {  
            var2 = "above 0";  
        } else {  
            var2 = "less than or equal to 0";  
        }  
        System.out.println(var2);  
    }  
}
```



Variable names and scope

- Usually it is illegal to have two variables in the same scope with the same name.

```
public class Point {  
    int x;  
    int y;  
    ...  
  
    public void setLocation(int newX, int newY) {  
        x = newX;  
        y = newY;  
    }  
}
```

- The parameters to `setLocation` are named `newX` and `newY` to be distinct from the object's fields `x` and `y`.

Variable shadowing

- An instance method parameter can have the same name as one of the object's fields:

```
// this is legal
public void setLocation(int x, int y) {
    ...
}
```

- Fields `x` and `y` are *shadowed* by parameters with same names.
- Any `setLocation` code that refers to `x` or `y` will use the parameter, not the field.

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.

'this' keyword

- Clarifies scope
- Means 'my object'

Usage:

```
class Example {  
    int memberVariable;  
    void setVariable(int newVal) {  
        this.memberVariable += newVal;  
    }  
}
```

Multiple constructors

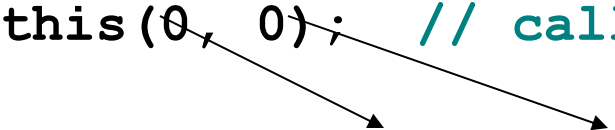
- It is legal to have more than one constructor in a class.
 - The constructors must accept different parameters.

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point() {  
        x = 0;  
        y = 0;  
    }  
  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    ...  
}
```

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

Packages

- Each class belongs to package
- Classes in the same package serve a similar purpose
- Packages are just directories
- Classes in other packages need to be imported
- Classes that are not explicitly put into a package are in the “default” package.

Packages

Defining Packages

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

Using Packages

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

Packages

- If the class is in a package named **test.pkg**,
 - then the first line of the source code will be
`package test.pkg;`
 - the source code file must be in a subdirectory named “pkg” inside a directory named “test”
`/test/pkg/ClassName.java`
 - Use “`javac test/pkg/ClassName.java`” to compile
 - Use “`java test.pkg.ClassName`” to execute

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`

Parameter Passing

- In general two ways in which an argument can be passed to a method
 - call-by-reference
 - changes made to the parameter will affect the argument used to call the method
 - call-by-value (Java)
 - changes made to the parameter of the method have no effect on the argument in the call

Java is Call-by-value

- what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
    /* This method causes no change to the arguments
       used in the call. */
    void noChange(int i, int j) {
        i = i + j;
        j = -j;
    }
}

class CallByValue {
    public static void main(String args[]) {
        Test ob = new Test();

        int a = 15, b = 20;

        System.out.println("a and b before call: " +
                           a + " " + b);

        ob.noChange(a, b);

        System.out.println("a and b after call: " +
                           a + " " + b);
    }
}
```

```
// Primitive types are passed by value.
class Test {
    /* This method causes no change to the arguments
       used in the call. */
    void noChange(int i, int j) {
        i = i + j;
        j = -j;
    }
}
```

```
class CallByValue {
    public static void main(String args[]) {
        Test ob = new Test();

        int a = 15, b = 20;

        System.out.println("a and b before call: " +
                           a + " " + b);

        ob.noChange(a, b);

        System.out.println("a and b after call: " +
                           a + " " + b);
    }
}
```


Objects as parameters

- When you create a variable of a class type, you are creating a reference to an object.
- When you pass object reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- Changes to the object inside the method do affect the object used as an argument.

```
// Objects are passed through their references.
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }
    /* Pass an object. Now, ob.a and ob.b in object
       used in the call will be changed. */
    void change(Test ob) {
        ob.a = ob.a + ob.b;
        ob.b = -ob.b;
    }
}

class PassObRef {
    public static void main(String args[]) {
        Test ob = new Test(15, 20);

        System.out.println("ob.a and ob.b before call: " +
                            ob.a + " " + ob.b);

        ob.change(ob);

        System.out.println("ob.a and ob.b after call: " +
                            ob.a + " " + ob.b);
    }
}
```

3

Method Overloading

- In Java, two or more methods within the same class can share the same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.

Method Overloading

```
// Demonstrate method overloading.
class Overload {
    void ovlDemo() { ←————— First version
        System.out.println("No parameters");
    }

    // Overload ovlDemo for one integer parameter.
    void ovlDemo(int a) { ←————— Second version
        System.out.println("One parameter: " + a);
    }

    // Overload ovlDemo for two integer parameters.
    int ovlDemo(int a, int b) { ←————— Third version
        System.out.println("Two parameters: " + a + " " + b);
        return a + b;
    }

    // Overload ovlDemo for two double parameters.
    double ovlDemo(double a, double b) { ←————— Fourth version
        System.out.println("Two double parameters: " +
                           a + " " + b);
        return a + b;
    }
}
```

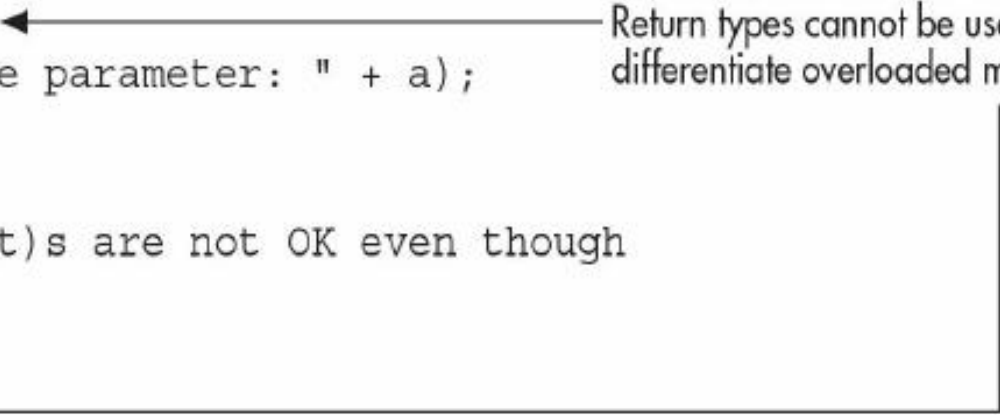
Method Overloading

```
class OverloadDemo {  
    public static void main(String args[]) {  
        Overload ob = new Overload();  
        int resI;  
        double resD;  
  
        // call all versions of ovlDemo()  
        ob.ovlDemo();  
        System.out.println();  
  
        ob.ovlDemo(2);  
        System.out.println();  
  
        resI = ob.ovlDemo(4, 6);  
        System.out.println("Result of ob.ovlDemo(4, 6): " +  
                           resI);  
        System.out.println();  
  
        resD = ob.ovlDemo(1.1, 2.32);  
        System.out.println("Result of ob.ovlDemo(1.1, 2.32): " +  
                           resD);  
    }  
}
```

Method Overloading

```
// One ovlDemo(int) is OK.  
void ovlDemo(int a) {  
    System.out.println("One parameter: " + a);  
}  
  
/* Error! Two ovlDemo(int)s are not OK even though  
   return types differ.  
*/  
int ovlDemo(int a) {  
    System.out.println("One parameter: " + a);  
    return a * a;  
}
```

Return types cannot be used to differentiate overloaded methods.



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 Constructors

// Demonstrate an overloaded constructor.

```
class MyClass {
```

```
    int x;
```

```
    MyClass() { ← Construct objects in a variety of ways.
        System.out.println("Inside MyClass() .");
        x = 0;
    }
```

```
    MyClass(int i) { ←
        System.out.println("Inside MyClass(int) .");
        x = i;
    }
```

```
    MyClass(double d) { ←
        System.out.println("Inside MyClass(double) .");
        x = (int) d;
    }
```

```
    MyClass(int i, int j) { ←
        System.out.println("Inside MyClass(int, int) .");
        x = i * j;
    }
```

```
}
```


Overloading Constructors

```
// Initialize one object with another.
class Summation {
    int sum;

    // Construct from an int.
    Summation(int num) {
        sum = 0;
        for(int i=1; i <= num; i++)
            sum += i;
    }

    // Construct from another object.
    Summation(Summation ob) { ←———— Construct one object from another.
        sum = ob.sum;
    }
}
```

Java API

Java API

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

Arrays with items

Create the array bigger than you need

Track the next “available” slot

```
Book[] books = new Book[10];
```

```
int nextIndex = 0;
```

```
books[nextIndex] = b;
```

```
nextIndex = nextIndex + 1;
```

Arrays with items

Create the array bigger than you need

Track the next “available” slot

```
Book[] books = new Book[10];
```

```
int nextIndex = 0;
```

```
books[nextIndex] = b;
```

```
nextIndex = nextIndex + 1;
```

What if the library expands?

ArrayList

Modifiable list

Internally implemented with arrays

Features

- Get/put items by index
- Add items
- Delete items
- Loop over all items

Array → ArrayList

```
Book[] books =  
    new Book[10];  
int nextIndex = 0;  
  
books[nextIndex] = b;  
nextIndex += 1;
```

```
ArrayList<Book> books  
= new ArrayList<Book>();  
  
books.add(b);
```

```
import java.util.ArrayList;
class ArrayListExample {
    public static void main(String[] arguments) {
        ArrayList<String> strings = new ArrayList<String>();
        strings.add("Evan");
        strings.add("Eugene");
        strings.add("Adam");

        System.out.println(strings.size());
        System.out.println(strings.get(0));
        System.out.println(strings.get(1));

        strings.set(0, "Goodbye");
        strings.remove(1);
        for (int i = 0; i < strings.size(); i++) {
            System.out.println(strings.get(i));
        }
        for (String s : strings) {
            System.out.println(s);
        }
    }
}
```


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

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

Inheritance

- A concept of hierarchical classification
- A Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food.
- A Red Delicious apple would, in turn, inherit all the qualities of all preceding classes, and would define only those qualities that make it unique.

Inheritance

- Without the use of hierarchies, each class would have to explicitly define all of its characteristics.
- Using inheritance, a class need only define those qualities that make it unique and it can inherit its general attributes from its parent.

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) {  
        ...  
    }  
}
```

Inheritance Basics

- Java supports inheritance by allowing one class to incorporate another class into its declaration.
- This is done by using the extends keyword. Thus, the subclass adds to (extends) the superclass.

```
public class Person {  
    String name;  
    char gender;  
    Date birthday;  
  
    int getAge(Date today) {  
        ...  
    }  
}
```

```
public class Student  
    extends Person {  
  
    ArrayList<Grade> grades;  
  
    double getGPA() {  
        ...  
    }  
}
```

```
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)

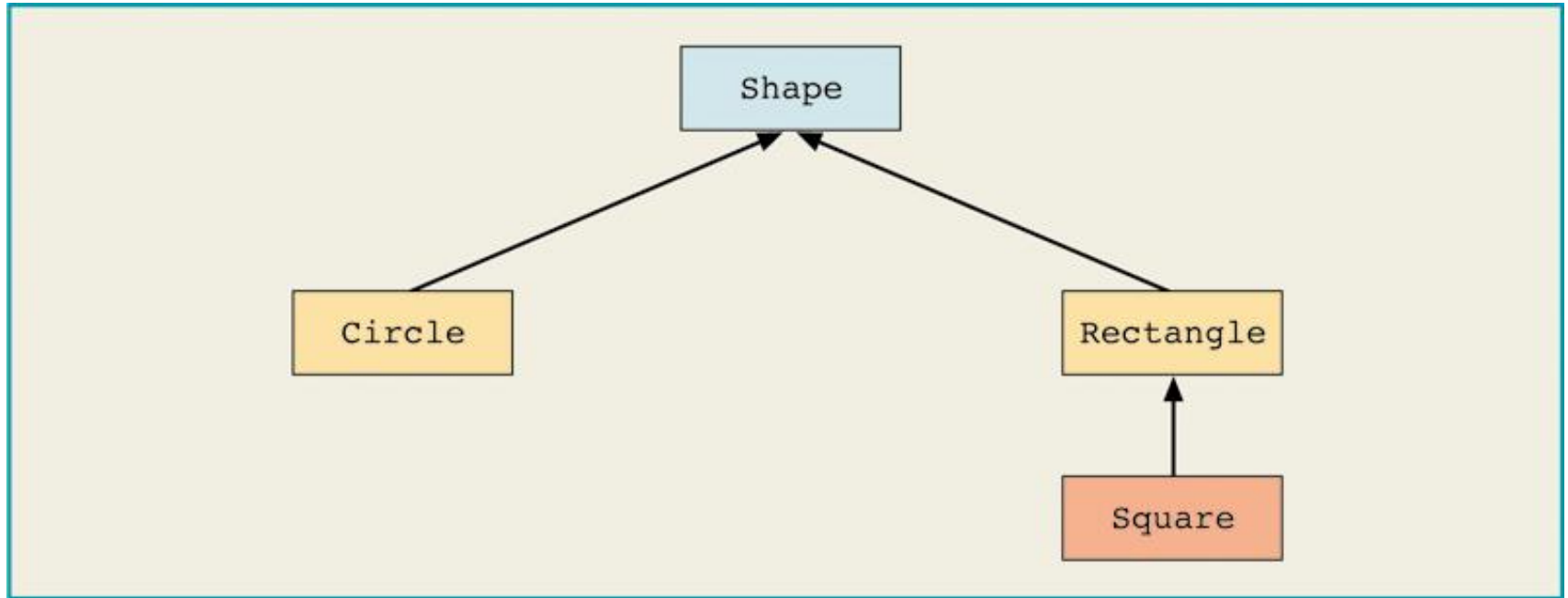


Figure 11-1 Inheritance hierarchy

```
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

```
public class Person {  
    String name;  
    char gender;  
    Date birthday;  
  
    int getAge(Date today) {  
        ...  
    }  
}
```

```
public class Student  
    extends Person {  
  
    ArrayList<Grade> grades;  
  
    double getGPA() {  
        ...  
    }  
}
```

```
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

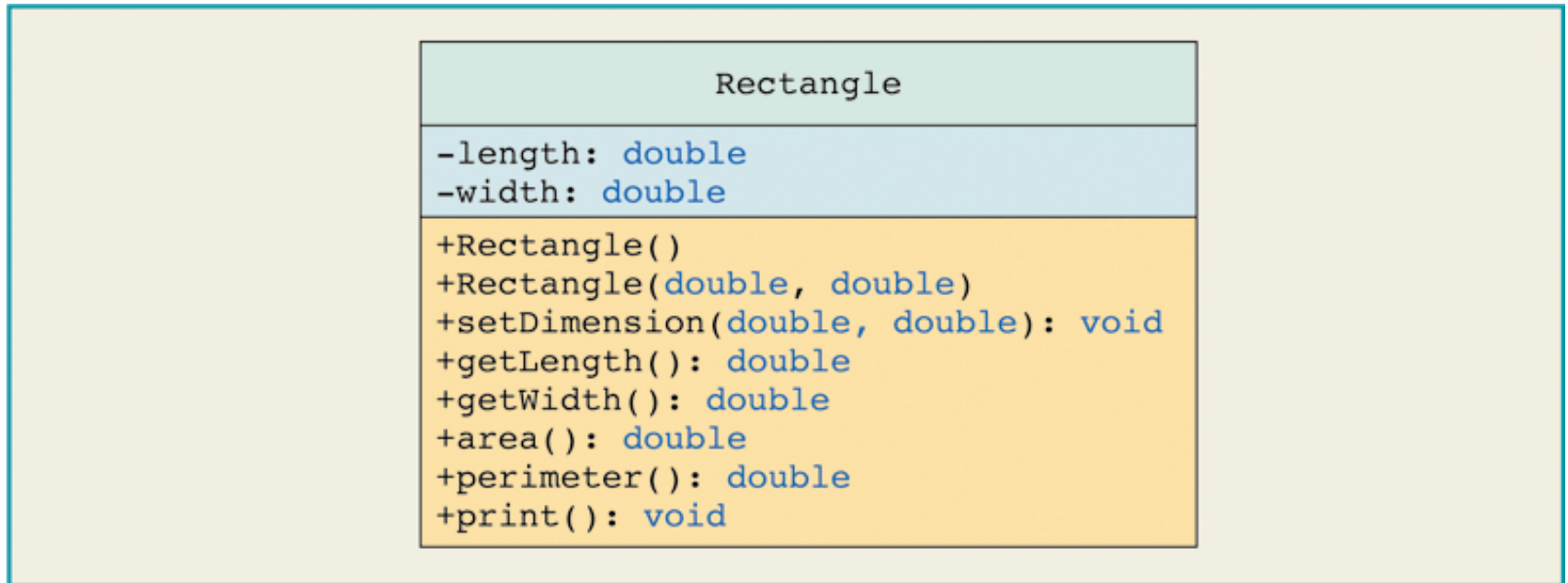


Figure 11-2 UML class diagram of the **class** Rectangle

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);
```

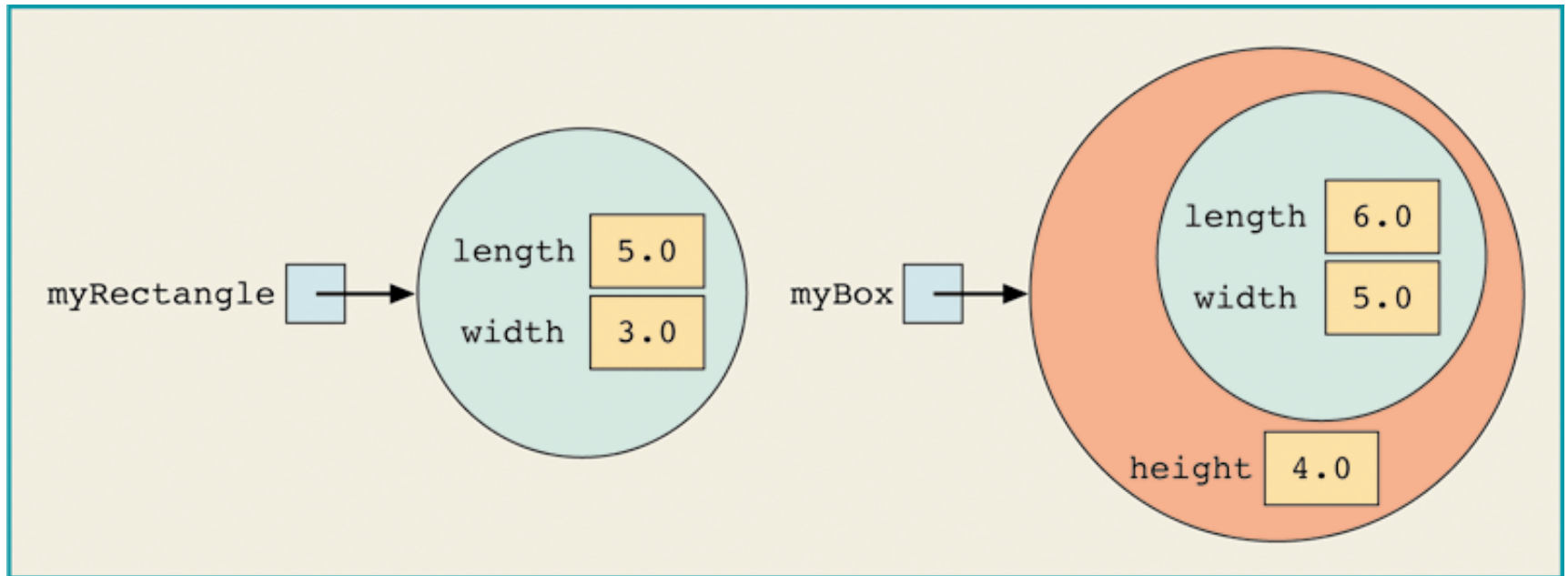


Figure 11-4 Objects myRectangle and myBox

UML Class Diagram: `class`

Box

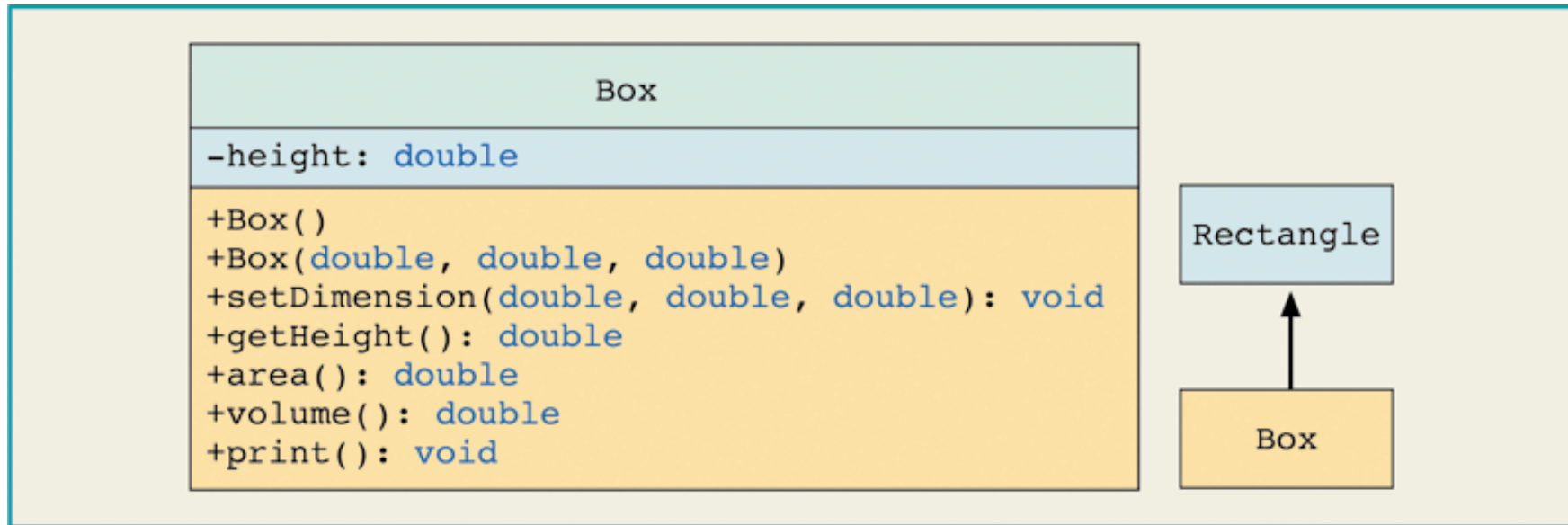


Figure 11-3 UML class diagram of the `class` `Box` and the inheritance hierarchy

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

Member Access and Inheritance

- Even though a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared private.
- Java programmers typically use accessor methods to provide access to the private members of a class.

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

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

```
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)

Constructors and Inheritance

- What constructor is responsible for building an object of the subclass
 - the one in the superclass,
 - the one in the subclass,
 - or both?
- The constructor for the superclass constructs the superclass portion of the object
- The constructor for the subclass constructs the subclass part

Constructors and Inheritance

- When only the subclass defines a constructor,
 - The superclass portion of the object is constructed automatically using its default constructor.
- When both the superclass and the subclass define constructors,
 - you must use Java's keywords, **super** to call superclass constructor

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;  
}
```

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

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://www.tiobe.com/tiobe-index/>
- Head First Java 2nd Edition
- Java a Beginner's guide 7th Edition