

Introduction to Objects and Object-Oriented Programming

Topics

– Introduction to Objects and Object-Oriented Programming

- Class
 - Design/blueprint of an object
 - Data
 - Methods
- Objects
 - Instance** of class.
- Encapsulation
 - Information Hiding
- Message passing
- Java Data Types.
- Fields
 - Static vs. Non static

Objects

- An object is a "thing," a "gizmo," a "gadget," ... an object.
- For example, a car, a soda machine, a dog, a person, a house, a bank account, a pair of dice, a deck of cards, a point in the plane, a TV, a VCR, an ATM machine, an elevator, a square, a circle, a flea, an elephant, a camera, a movie star, a computer mouse, a live mouse, a phone, an airplane, a song, ... just about anything is an object.
- In computing, a window is an object, so is a mouse, a menu, a textbox, and a button.
- Objects come in all shapes, sizes, and colors. An object may be physical, like a radio, or intangible, like a song.
- For our purposes, however, objects are entities that have
 1. attributes, (characteristics or properties), and
 2. methods, (actions or behaviors of an object).

Elevator Object

- Notice that the three elevator objects have different attribute *values*.

The attribute values determine the *state* of an object.

- Thus the state of elevator 1 is that the floor is 3 and the door is open.
- The state of elevator 3 is that the current floor is 2 and the door is closed.
- **Each elevator object has a unique state.**
- On the other hand, **all elevator objects have the same behavior.**
- However, all three objects can *do* the same things (open the door, close the door etc.).
- **All three have the same methods.**

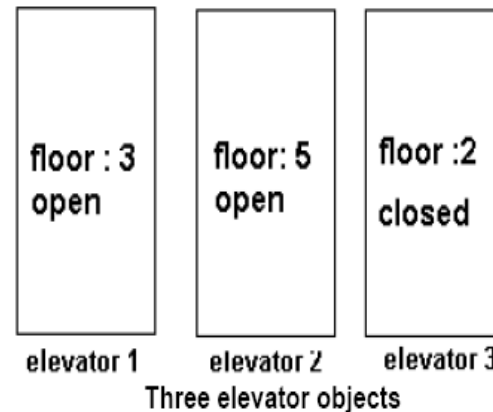
Example 1: An **elevator** is an object.

The attributes? Perhaps:

1. the current floor
2. whether or not the door is open or closed

The methods (actions/behaviors) might be:

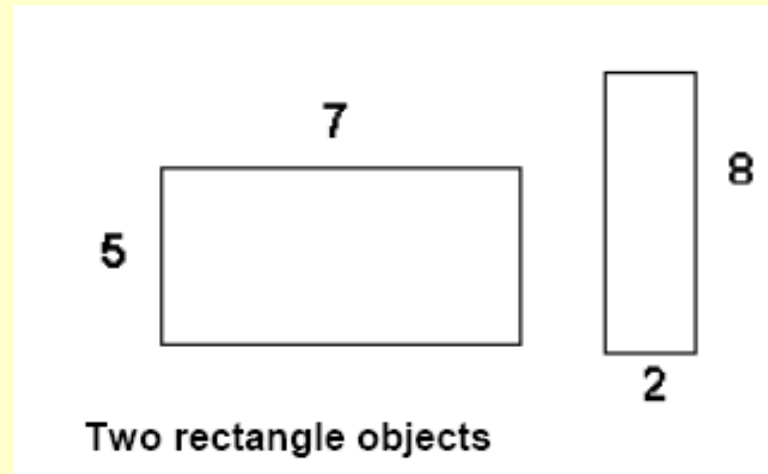
1. give (display) the current floor
2. open the door
3. close the door
4. change the current floor
5. ring the alarm



Rectangle Object

- A **rectangle** is an object
- Some possible properties or **attributes** of a **rectangle** are:
 1. length and
 2. width
- Some possible methods are:
 1. get the length
 2. get the width
 3. change the length
 4. change the width
 5. get the area
 6. get the perimeter

Here are two rectangle objects:



- The state of the **first** rectangle object is $\{\text{length} = 7, \text{width} = 5\}$;
- The state of the **second** rectangle is $\{\text{length} = 2, \text{width} = 8\}$.

Computer Window Object!

- A **computer window** is an object
 - Some of the (many) attributes include:
 1. length
 2. width
 3. background color
 4. font style
 5. font color
 6. state – maximized, minimized or downscaled
 - Some of the (many) methods include:
 1. resize the window (change length and width)
 2. maximize
 3. minimize
 4. change background color
 5. change font etc.

Defining Object

- **In the context of a computer program**, you might think of an *object* as a **representation, model** or abstraction of **some entity** *consisting of*
 1. **data** (attributes) and
 2. **functions (methods)** which use or manipulate the data
- An object's *data* determines its **state**. For example, the data for the first elevator (above) specifies that the “floor” attribute has value 3 and the door is open. The current state of the first rectangle indicates that the length of the rectangle is 7 and the width is 5.
- The **methods/functions specify what an object does**, i.e., the **behavior** of an object.

How do you define Object?

- The attributes and methods of an object depend on our specific use and view of an object.
- For example, in one application, a rectangle object might be a simple geometrical figure with just two attributes, length and width.
- Another, perhaps graphical, view of a rectangle might include color and location (x and y coordinates) among the attributes.
- Similarly, an elevator has many potential attributes (carpet color, number of passengers, maximum weight, , date of last inspection) but only a few attributes are of interest in any application.

Object has Object!

- **Some attributes themselves might be other objects.**
- For example, a light object may have:
 - attributes:
 - number of watts
 - current state (on or off)
 - methods:
 - turn light on
 - turn light off
- **Now a light object may be part of (an attribute of) an elevator object.**

Your Turn!

A circle is an object.

| | |
|---|--|
| <ul style="list-style-type: none">•attribute (data) of a circle:<ul style="list-style-type: none">–radius, a real number. | <ul style="list-style-type: none">•The methods (functions) might be<ol style="list-style-type: none">a. give (return) its areab. give (return) its circumference. |
|---|--|

In each of the examples, the attributes and methods have been chosen arbitrarily. Indeed, choosing the “right” attributes and methods for an object is a skill and an art that comes with practice and patience.

Lets Try another one!

A Bank account is an object.

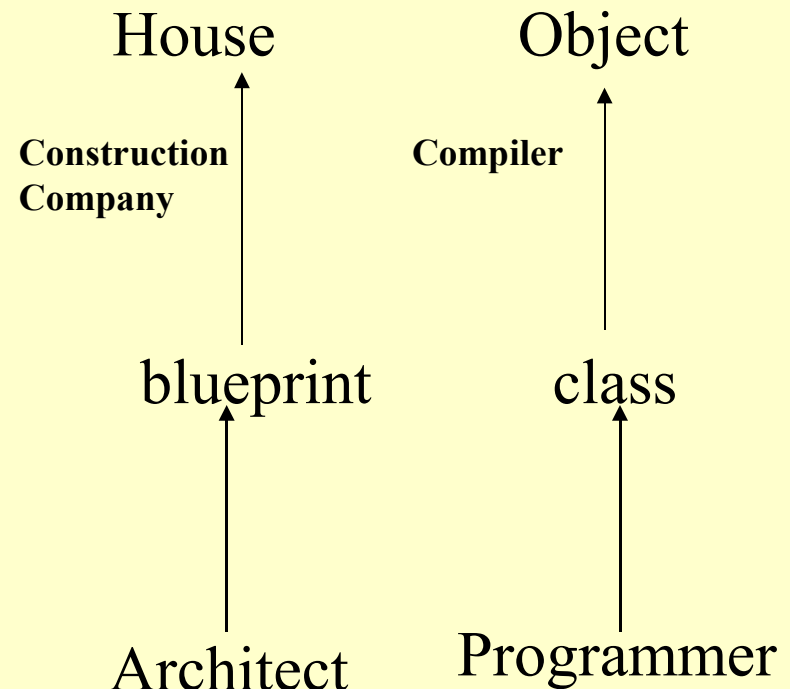
| data or attributes of a bank account might be | The methods/functions/operations might be |
|--|---|
| a. an ID number b. customer's name, c. address, and d. balance. | a. give (return) the balance b. give personal information about the account owner c. make a deposit d. make a withdrawal |

What is Classes?

- A *class* is a **template, blueprint, or description of a group of objects.**
- Every object is described by some class. For example,
- An elevator class specifies the characteristics and behaviors of all elevator objects. The elevator class is a general *description* of an elevator. **An elevator class is *not* an elevator.**
- A rectangle class describes the attributes and methods of all rectangle objects. A rectangle class may specify that every rectangle object has both a length and a width. However, **a rectangle class is not a rectangle.**

Analogy

- An **architect's blueprint** is analogous to a **class**.
- A **blueprint is not a house** but a **description or specification of a potential house**.
- When a **builder constructs** two real houses from a **blueprint**, well, *now* we have two "house objects."
- The **skill of the programmer in defining classes** is akin to the **skill of the architect**, and the **labor of the compiler in building objects** is like the **work of the construction company**.



Object Analogy

An object is an instance of a class!

- Just as a builder creates houses from a blueprint,
- From one blueprint, a builder can build many houses.
- Every house builds from the blue print
- a program creates *objects* from a class.
- From one class, a program can create many objects
- Every object is “manufactured” according to some class specification.
- Every object belongs to some class.

Class Example

- Let us look at an example of a Rectangle class in Java.
- The class is a description (in Java) of the attributes and behaviors of a rectangle object.
- For now, don't be concerned with the syntax or any of the Java particulars.

Rectangle Class

```
public class Rectangle
{
    //Every Rectangle object has both length and width
    attributes (int)
    private int length;
    private int width;
    //default values for a rectangle object are length =
    1 and width = 1
    public Rectangle() // default constructor
    {
        length = 1;
        width = 1;
    }
    //can create a Rectangle object with any
    dimensions
    public Rectangle(int x,int y) //constructor
    {
        length = x;
        width = y;
    }
}
```

```
//can change the dimensions of any rectangle
object
public void changeDimensions(int x,int y)
    // mutator
    {
        length = x;
        width = y;
    }
//gives the area of a rectangle object // accessor
public int getArea()
    {
        return length*width;
    }
//gives the perimeter of a rectangle object
public int getPerimeter()
    {
        return 2*(length+width);
    }
}
```

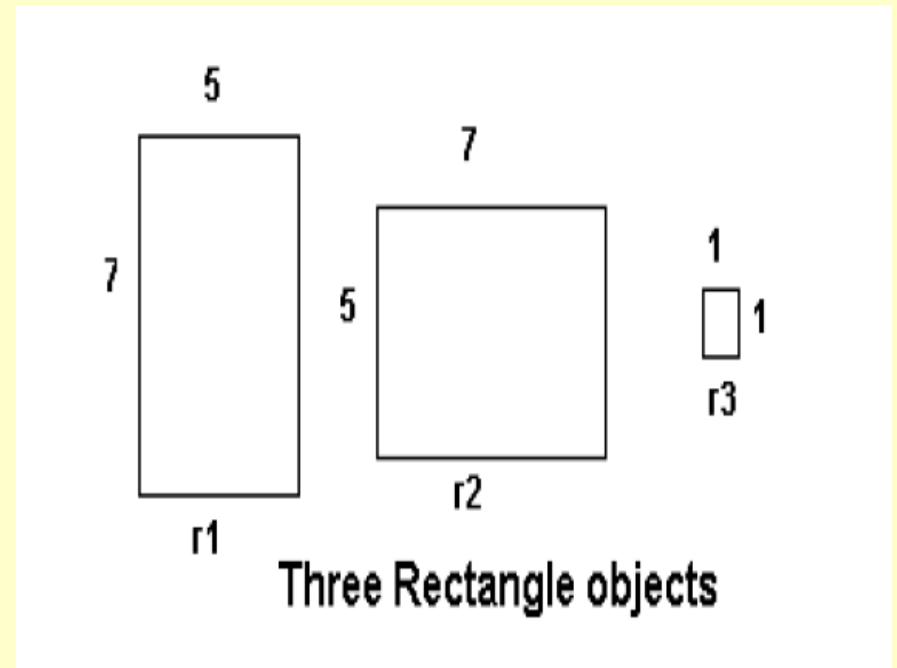
Analogy of the Code

- The preceding code (a Rectangle class) is a *template* for a rectangle. According to the specifications, any potential rectangle object has both **a length and a width** of type int. Moreover any rectangle object can
 - **change its dimensions,**
 - **give its area, and**
 - **give its perimeter.**
- OK, we know what a rectangle object has and what it can do. So, we can now manufacture, create, instantiate as many rectangles as we like. We have the blueprint, so let's start production:

```
// makes a 5 X 7 rectangle named r1  
Rectangle r1 = new Rectangle(5,7);  
// makes a 7 X 5 rectangle named r2  
Rectangle r2 = new Rectangle(7,5);  
// makes a default 1 X 1 rectangle named r3  
Rectangle r3 = new Rectangle();
```

Object

- With three magic statements, we have created three rectangle objects --- built according to the specifications of our class/blueprint.
- Each rectangle object has a length property and a width property with appropriate values:



OOP Concept: Encapsulation

- Webster defines encapsulation as being “enclosed by a capsule.”
- Real world examples of encapsulation surround us:
- A computer is an example of real world encapsulation. The chips, boards, and wires of a computer are never exposed to a user. Like the TV viewer, a computer user operates a computer via an interface -- a keyboard, screen, and pointing device.

Example: Encapsulation

- A cabinet hides (encapsulates) the “guts” of a television, concealing from TV viewers the internal apparatus of the TV.
- Moreover, the television manufacturer provides users with an *interface* --the buttons on the TV or perhaps a remote control unit.
- To operate the TV and watch *The Simpsons* or *Masterpiece Theatre*, viewers utilize this interface. The inner circuitry of a TV is of no concern to most TV viewers.
- Cameras, vending machines, slot machines, DVD players, lamps, cars, video games, clocks, and even hourglasses are all physical examples of encapsulation.

Encapsulation: Explained

- Each of the devices enumerated above supplies the user with an interface — switches, buttons, remote controls, whatever -- for operation.
- Technical details are tucked away and hidden from users.
- Each encapsulated item functions perfectly well as a “black box.”
- Certainly, Joe User need not understand *how* his Radio Shack gadget is constructed in order to operate it correctly.
- A user-friendly interface and perhaps an instruction manual will suffice.

Encapsulation: Explained

- Encapsulation has a similar (though somewhat expanded) meaning when applied to software development and object oriented programming:

The ability to provide users with a well-defined interface to a set of functions in a way which hides their internal workings. In object-oriented programming, the technique of keeping together data structures and the methods (procedures) which act on them.

– The Online Dictionary of Computing

- Java provides encapsulation, as defined above, via classes and objects.
- As we have already seen, classes bundle data and methods into a single unit. *Classes encapsulate.*

Rectangle Class: Revisited

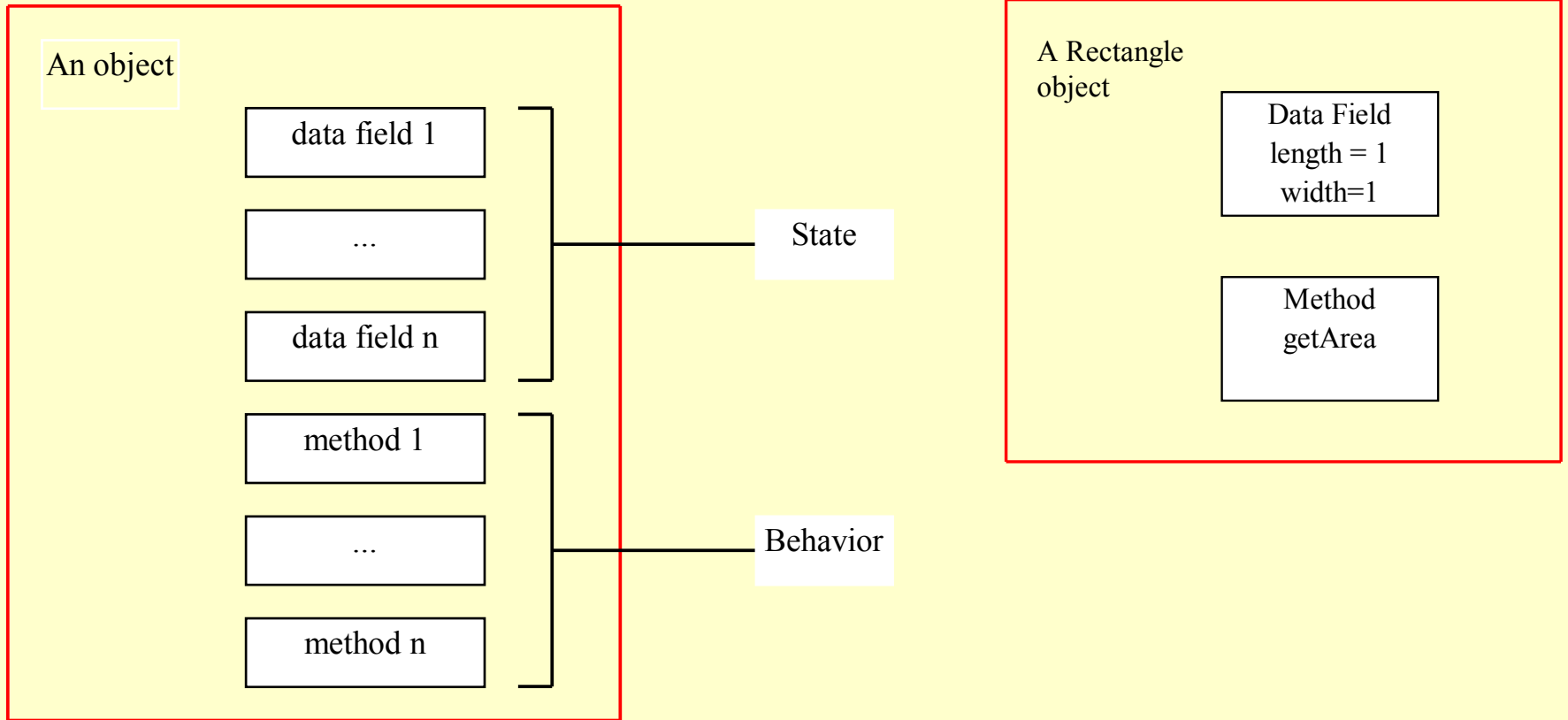
```
public class Rectangle
{
//Every Rectangle object has both length and width
attributes (int)
private int length;
private int width;
//default values for a rectangle object are length = 1
and width = 1
public Rectangle() // default constructor
{
length = 1;
width = 1;
}
//can create a Rectangle object with any dimensions
public Rectangle(int x,int y) //constructor
{
length = x;
width = y;
}
```

```
//can change the dimensions of any rectangle
object
public void changeDimensions(int x,int y) //
mutator
{
length = x;
width = y;
}
//gives the area of a rectangle object // accessor
public int getArea()
{
return length*width;
}
//gives the perimeter of a rectangle object
public int getPerimeter()
{
return 2*(length+width);
}
}
```

Rectangle: Test Driver!

```
public TestRectangle{
public static void main(String args[])
    {// makes a 5 X 7 rectangle named r1
      Rectangle r1 = new Rectangle(5,7);
      // makes a 7 X 5 rectangle named r2
      Rectangle r2 = new Rectangle(7,5);
      //makes a default 1 X 1 rectangle named r3
      Rectangle r3 = new Rectangle();
      System.out.println(r1.getPerimeter() );
        double area = r2.getArea();
        r3.changeDimensions(7,3);
    }
}
```

OO Programming Concepts



Encapsulation

- The Rectangle class provides a simple example of encapsulation – data and methods, attributes and functionality, are combined into a single unit, a single class.
- Furthermore, all data are accessed not directly but through the class methods, i.e. via an interface /TestRectangle.
- The user of Rectangle – the client -- need not know *how* the class is implemented – only how to *use* the class.
- Variable names are of no concern to the client. If the client wishes to know the perimeter of a Rectangle object, the interface provides an accessor method.

Encapsulation

- If the client wants to dimension the size of a Rectangle object, the client simply uses the mutator method available through the interface.
- That the length of the sides of a square is held in a variable called *dimension* is irrelevant to the client.
- Further, if the implementation of the class is modified, programs utilizing the Rectangle class will not be affected provided the interface remains unchanged.
- Like a TV or a camera, the inner workings of the Rectangle class are encapsulated and hidden from the client.
- Public methods provide the interface just as the remote control unit provides the interface for a TV viewer.

Encapsulation: Information Hiding

- Another term that is often associated with encapsulation is *information hiding*.
- Many authors regard encapsulation and information hiding as synonyms.
- However OO purists might define **encapsulation** as the *language feature* allowing the bundling of data and methods into one unit
- and **information hiding** as the *design principle* that restricts clients from the inner workings of a class.
- With this distinction, programs can have encapsulation without information hiding.
- Regardless of how you define your terms, classes should be designed with a well-defined interface in which implementation decisions and details are hidden from the client.

Messages

- In a Java program, objects interact with other objects by sending messages.
- Messages are similar to function calls in procedural style programming.
- The following three statements send messages to r1, r2 and r3, respectively:

```
System.out.println(r1.getPerimeter() );  
area = r2.getArea();  
r3.changeDimensions(7,3);
```

- The purpose of the messages should be pretty obvious:
 - “r1, get your perimeter!”
 - “r2, get your area!”
 - “r3, change your dimensions!”

Example of Message Passing

- From our perspective a Dog object is a very simple creature:
- Our simple Dog class which has only a single attribute:
 - bark.
- A Dog object can do only two things:
 - set its bark and
 - speak, i.e., bark

```
public class Dog
{
    /*A Dog object has but a single
    attribute—its bark*/
    private String bark;
    /*set the sound: "woof-woof," "bow-
    wow" etc.*/
    public void setBark(String s) {
        bark = s;
    }
    /* Every dog can bark*/
    public void speak() {
        System.out.print(bark);
    }
}
```

Creation of “Fido” and “Brutus”

- We now **create (instantiate)** a few Dog objects:

```
/* create a Dog named fido*/
```

```
Dog fido = new Dog();
```

```
/*create a Dog named brutus*/
```

```
Dog brutus = new Dog();
```

- send a few messages** to the critters:

```
/* a message to fido*/
```

```
fido.setBark(“Bow-Wow”);
```

```
/* a message to brutus*/
```

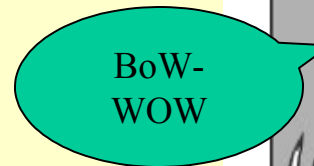
```
brutus.setBark(“Woof-Woof”);
```

```
/* fido, speak, boy!!*/
```

```
fido.speak();
```

```
/* you too, brutus, speak*/
```

```
brutus.speak();
```



Java Data Types

- Reference Types
- Primitive Data Types

Primitive Data Types

- The usual suspects:
 - byte, short, int, long
 - char
 - boolean
 - float, double
- The usual operators and rules apply mostly

Reference Types (1)

- Can't write about objects without referring to them
- Reference value: the only way to refer to an object
- Java has:
 - reference values (or just *references*)
 - reference expressions
 - reference variables
 - assignment of references

Declarations, Variables, Assignments

- C-like rules apply for the most part
- Type specifier followed by identifier list

Objects

- Only accessed via reference values
 - `Rectangle r1 = new Rectangle();`
- Not seen— we (the programmers) only get references to them and send them messages.
 - `r1.getArea();`

Objects Can NOT Be

- assigned to a variable
 - ~~int x = Rectangle new Rectangle();~~
- the value of an expression
 - new Rectangle() = x+y;
- passed as an argument in a message to an object
 - r1.setArea(new Rectangle());
- returned by an object responding to a message
- declared

But References Can Be

- assigned to a variable
- the value of an expression
- passed as an argument in a message to an object
- returned by an object responding to a message
- declared

Different Reference Types

- References to objects of different classes are different types

Every class implicitly defines a distinct reference type

References Are Not Pointers

- can't take "address" of object— can not refer to object without reference
- can't do arithmetic with references
- Java has no pointers

What's The Use Of References?

- only way to access an object
- only way to send a message to an object

Messages: Sending

- Message Form:

methodName(argument1, argument2, ...,
argumentN)

- Sending a Message (Form):

reference • Message (i.e. r1.getArea();)

Messages: Response From Receiver

- value
 - send message form can be used in expression
 - often the right side of an assignment statement
 - can be primitive data OR a reference to an object
- void

Java Pre-defined Classes

- Huge number of predefined classes
 - utility, I/O, GUI, network, time/day, database, math, etc.
- `PrintStream` class:
 - `println(string)`, `print(string)`
- `String` class:
 - `toUpperCase`, `trim`, `substring`, `indexOf`, etc.
- `File` class
 - `delete`, `renameTo`

Java Pre-defined Objects

- System.out, System.err, System.in
- String constants

Sample Code 1

```
String    s;  
int       i=0, k;  
s = "hello, world";  
k = s.length();  
while (i<k-1) {  
    System.out.println(s.substring(0,i+1));  
    i++;  
}
```

Cascading Messages

```
String    s;  
s = "Hello".concat(" World").toUpperCase();  
System.out.println(s);
```

Creating Objects

- `new` keyword
- constructor + arguments
- expression's value is reference to create object

Creating And Using An Object: Example

```
File junk;  
junk = new File("garbage");  
junk.delete();
```

Class Definitions

```
class NameOfClass {  
    method definitions (including constructors)  
    instance variable declarations  
}
```

Method Definitions

- similar to function definitions
- additional keywords: public or private
- optional keyword: static
- optional phrase: throws
SomeKindOfException

Class Definition Example 1

```
class Laugher4 {  
    public Laugher4(String defaultSyl) {  
        default = defaultSyl;  
    }  
  
    public void laugh() {  
        System.out.println(default);  
    }  
  
    private String default;  
}
```

Class Definition Example 2

```
class Laugher2 {
    public Laugher2() { default = "ha"; }

    public Laugher2(String defaultSyl) {
        default = defaultSyl;
    }

    public void laugh() {
        System.out.println(default);
    }

    public void laugh(String syl) {
        System.out.println(syl);
    }
    private    String default;
}
```

Signatures and Overloading

- Signature: method name + argument types
- Overloading: methods of the same name but different signatures

```
public Laughter2() {  
public Laughter2(String defaultSyl) {  
public void laugh() {  
public void laugh(String syl) {
```

Input and Output

- Input and output classes model different i/o services
- Setting up i/o involves a composition of constructors
- Example (input):

```
BufferedReader br = new BufferedReader(  
    new InputStreamReader(  
        new FileInputStream(  
            new File("USData"))));  
  
    // file input
```

BufferedReader

- Provides a `readLine` method
- Returns reference to `String` modeling line read in OR returns null if end of file is reached
- Setting up i/o involves a composition of constructors
- Example (input):

```
String s = br.readLine();
while (s!=null) {
    System.out.println(s);
    s = br.readLine();
}
```

Command Line Input

```
/*saved as Prog4.java*/
public class Prog4 {
public static void main
    (String args[])
{
System.out.println(args[0]);
System.out.println(args[1]);
}
}
```

- If you run this program with the command
> java Prog4 Dopey Grumpy
- the two strings entered at the command line, “Dopey” and “Grumpy,” **are stored in the array args.**
- Consequently, args[0] holds the string “Dopey” and args[1] holds “Grumpy.”
- **Notice that arrays are indexed from 0, as they are in C++.**
- **Output:**
Dopey
Grumpy

Input/Output Trial!

- Write a program that will take one input, teacher's last name. Then display it back for the user with a welcome message. So, if we run the program with the command:

Prog Name  `>java tecProg2 maria` input

The program will display back:

Output:

Hello Maria, Welcome to Tec at Brooklyn College!

Keyboard Input!

- System.in is an InputStream
- Make sure to include:
 - import java.io.*;
- By default Java includes Java.lang package.
- Example:
- ```
// keyboard input
BufferedReader keyboard =
new BufferedReader(
 new InputStreamReader(
 System.in));
```

```
/*saved as Prog5.java*/
import java.io.*;
public class Prog5 {
public static void main (String args[])
{
// keyboard input
System.out.println("Type your name");
BufferedReader keyboard = new
 BufferedReader(new
 InputStreamReader(System.in));
String name= keyboard.readLine();
System.out.println("your name is" +name);
}
}
```

# A Name Class

```
class Name {
 private String first, last, title;

 public Name(String first, String last) {
 this.first = first;
 this.last = last;
 }

 public String getInitials() {
 String s; // M Azhar
 s = first.substring(0,1); // M
 s = s.concat("."); //M.
 s = s.concat(last.substring(0,1)); //M.A
 s = s.concat("."); //M.A
 return s;
 }
}
```

# Name Class (2)

```
public String getLastFirst() {
 return lastName.concat(", ").concat(firstName);
}
```

```
public String getFirstLast() {
 return first.concat(" ").concat(last);
}
```

```
public void setTitle(String newTitle) {
 title = newTitle;
}
```

# Name Class (3)

```
public static Name read(BufferedReader br)
 throws Exception {
 String first, last;
 first = br.readLine();
 last = br.readLine();
 return new Name(first, last);
}
}
```

# Using The Name Class

```
public class TestName{
public static void main(String [] args)
{
BufferedReader br = new BufferedReader(new
 InputStreamReader(
 System.in));
 Name n;
 n = Name.read(br);
 System.out.println(n.getInitials());
}
}
```

# Class Variables: Static

- Although a class is not an object, a class can have **both** class variables and class methods.
- Class variables and class methods can exist whether or not any object is created.
- Class variables and class methods are indicated with the keyword **static**.

# Static (Class) Methods

A class/static method, as we have already seen, is a method that:

- exists as a member of a class,
- may be invoked using either the class name or the name of an object,
- may not access instance variables, (Since class methods may be invoked regardless of whether or not any object have been created, object/instance variables cannot be accessed by static methods.)
- is often used when it is important to know how many class instances exist or to restrict the number of instances of a class.

# Static (Class) Variables

If a class contains a static variable then:

- ∇ • All objects/instances of the class *share* that variable.
- ∇ • There is only one version of the variable defined for the whole class.
- ∇ • The variable belongs to the class.
- ∇ • The variable exists regardless of whether or not any objects have been created.
- ∇ • The variable may be accessed using either the class name or an object name, if an object has been created.

## **Example:**

- Static or class variables are often used for constants. The Math class contains two class constants: Math.PI and Math.E ( approximate value: 2.71828).

# Static

- The Java System library contains many class/static methods. The methods of Math are all static as are the (final) variables.
- Of course, allowing static methods and variables is contrary to the principles of object-oriented programming since Java is providing a mechanism for what amounts to global variables and methods!

# Static

- The following simple class contains two class/static variables:
  - The variable *count* keeps track of the number of Circle objects that have been created. One version of count exists for the entire class. All objects access this same variable.
  - The static variable pi is a constant. Notice the keyword *final*. Also note that pi is declared public so that any other class may access it as Circle.pi.

# Example

```
public class Circle
{
 static private int count = 0; //class variable
 public final static double pi = 3.14159; //class variable
 private double radius;

 //constructors
 public Circle() //default constructor
 {
 radius = 0;
 count++;
 }
 public Circle(double radius) //one argument constructor
 {
 this.radius = radius; //this.radius designates variable from the class
 count++;
 }

 public double area()
 {
 return pi*radius*radius;
 }

 public static int getCount() // this is a CLASS method
 {
 return count;
 }
}
```

# Test Driver

The following small class uses both the static data and methods of Circle.

```
public class Test
{
 public static void main(String args[])
 {
 System.out.println(" "+Circle.pi+ " "+ Circle.getCount());
 }
}
```

The output from this class is:

```
3.14159 0
```

# Explanation

- The class method, `getCount()`, returns the number of `Circle` objects that have been created.
- Notice that `getCount()` accesses `count` – a class/static variable. Remember: **a static method cannot access an instance variable, since instance variables may not even exist.**
- The example also illustrates use of the keyword *this*. Sometimes the parameter has the same name as one of the instance variables.
- To differentiate between the parameter and the private variable, use the keyword *this*. As in C++, *this* is a reference to the invoking object.

# Functions -- Static Methods

```
class Max {
 public static void main(String[] args){
 int[] a = new int[]{5,6,1,2,7,9,0,10};

 System.out.println(" max="+max(a));
 }
}
```

```
 static int max(int[] a){
 int max = a[0];
 for (int i=1; i<a.length; i++){
 if (a[i]>max) max = a[i];
 }
 return max;
 }
}
```

# Functions -- Static Methods

```
class PrintArray {
 public static void main(String[] args){
 int[][] a = new int[][]{{1,2,3}, {4,5,6}, {7,8,9}};
 printArray(a);
 }
 static void printArray(int[][] a){
 for (int i=0; i<a.length; i++){
 for (int j=0; j< a[0].length; j++){
 System.out.print(a[i][j]+" ");
 }
 System.out.println();
 }
 }
}
```

# View Classes as Modules

```
class Sort {
 public static void selectionSort(int a[]) {
 int tmp;
 for (int i=0; i<a.length; i++)
 for (int j=i+1; j<a.length; j++)
 if (a[i]>a[j]) {
 tmp = a[i];
 a[i] = a[j];
 a[j] = tmp;
 }
 }
 }
}
```

# View Classes as Modules

```
public class TestSort {
 public static void main(String[] args){
 int[] a = new int[]{5,6,1,2,7,9,0,10};
 Sort.selectionSort(a);
 for (int i=0; i<a.length; i++){
 System.out.print(a[i]+" ");
 }
 }
}
```