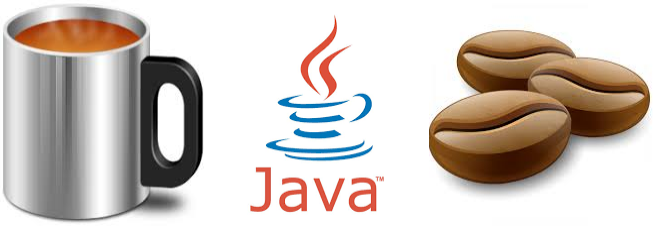
CSC3301 Java Workshop



Mohammed Hassan
mhassan.se@buk.edu.ng

Classes and Objects
Introduction to Inheritance

Week 2 &3

- Classes and Objects (Chapter 4)
  - Classes
  - Constructors
  - Methods
  - Creating Objects
  - Using Objects
- Inheritance Part 1 (Chapter 5)
  - Inheritance: Key Definitions
  - The Purpose of Inheritance
  - An Example of Inheritance
  - What You Can Do in a Subclass
  - Private Members in a Superclass
  - Casting Objects
  - Overriding Methods
  - Hiding Fields
  - Constructor Chaining

Contents

- Classes
- Constructors
- Methods
- Creating Objects
- Using Objects

Declaring Classes

- A class declaration consists of the *class* keyword, a class *name* and its *body*:

```
class ClassName {
    fields
    constructors
    methods
}
```

class body

- A class may or may not declare any of the three components of its body.

Implementation of a Bicycle, ver. 2

```
// File: Bicycle.java
public class Bicycle { // Class declaration
    private int cadence; // Fields
    private int gear;
    private int speed;
    public Bicycle(int startCadence, int
    startSpeed, int startGear) {
                        //  Constructor
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }
    public int getCadence() { // Methods
        return cadence;
    }
    public void setCadence(int
    newValue) {   cadence = newValue;
    }
    public int getGear() {
        return gear;
    }
    public void setGear(int newValue) {
        gear = newValue;
    }
    public int getSpeed() {
        return speed;
    }
    public void applyBrake(int decrement) {
        speed -= decrement;
    }
    public void speedUp(int increment) {
        speed += increment;
    }
} // End of class declaration
```


Comments on the Previous Slide

- The modifier *public* determines that other classes can access Bicycle:
  - `public class Bicycle { // Class declaration`
- It is common to make fields *private*. This means that they can only be *directly* accessed from the *Bicycle* class:
  - `private int cadence; // Fields`
- We can access the field *indirectly* by adding public methods that obtain the field values for us.
  - An example:

```
public void setCadence(int newValue){
        cadence = newValue;
}
```

Overloading Methods

- Methods can be overloaded.
- Methods within a class can have the same name, but a different parameter list.
- Overloaded methods are differentiated by the number and the type of the arguments passed into the method.
- One cannot declare more that 1 method with the same name and the same parameter list.
- The compiler does not consider return type and modifiers when differentiating methods.

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```


Example: Overloading Methods

- java.io.PrintStream defines the following overloaded methods (System.out. …):

void println()	terminates the current line by writing the line separator string
void println(float x)	prints a float and then terminates the line
void println(int x)	prints an integer and then terminates the line
void println(Object x)	prints an Object and then terminates the line
void println(String x)	prints a String and then terminates the line

Constructors

- Constructors are special methods of a class.
- Constructors have the same name as the class.
- Constructors are invoked to create and initialize objects.
- A class may or may not define a constructor.
- The ***default*** *constructor* has no arguments.
- A **default** constructor is automatically created if the class defines no constructors.

```
// See slide 14
public Bicycle(int startCadence,
    int startSpeed, int startGear){
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}

// a no-argument constructor
// default constructor
public Bicycle() {
    gear = 1;
    cadence = 10;
    speed = 0;
}
```

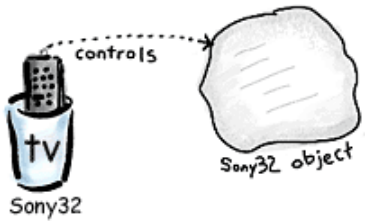

Example: Overloading Constructors

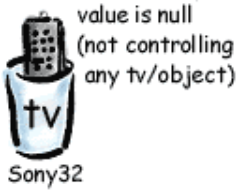
- java.lang.String defines these constructors:

String()	A String object that is an empty character sequence
String(byte[] value, Charset charset)	Creates a new String by decoding the **value** array using the specified **charset**
String(char[] value)	A new String that is the sequence of characters currently contained in **value**
String(String original)	Creates a String that is a copy of the **original** String
String(StringBuffer buffer)	Creates a new String from the **buffer** argument

Creating Objects

- In Java, **objects are created on the heap**, and a REFERENCE to the object is stored as a value of the variable (in the cup, as shown in picture). Think of it as a remote control to a specific type of object.
- At the café customers may say: "I'd like a reference to a new Sony32 television please, and name it TV." which in Java:
  - Sony32 tv = new Sony32();
- If you say: "declare but don't initialize with an actual Sony32 object"
  - Sony32 tv;



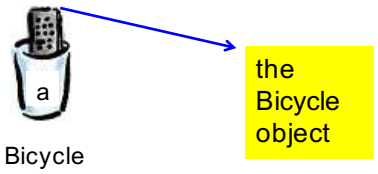


Creating Objects

- A class provides the blueprint for objects, i.e., an object is created from a class.
- For example:
  - String os = new String("Name");
  - Bicycle bike1 = new Bicycle(); // Lecture 1, Slide 42
- Each of these statements has 3 parts:
  - **Declaration** – the code on the left side of the assignment operator;
  - **Instantiation** – the keyword *new* is a Java operator that creates the object;
  - **Initialization** – the keyword *new* invokes the constructor that initializes the new object.

Instantiating a Class

- An **object** is the **instance** of a class.
- To instantiate a class is the same as to create an object.
- The *new* operator instantiates a class by allocating memory for a new object and returning a reference to that memory.
  - Bicycle bike = new Bicycle();



Declaring a Variable to Refer to an Object

- To declare a variable, one writes a statement:
  - type name;
- This statement associates *name* with its *type*.
- With a primitive variable, this declaration also reserves the memory to store any allowed value of the variable.
- With an object reference, the reserved memory can only store a reference, i.e., declaring a reference variable does not create an object:
  - String str; // does not create a String object

Steps to Create an Object

- Objects are created by invoking the operator *new* on a class.
- The keyword *new* is followed by a constructor of the class to be instantiated.
- A constructor is invoked after the fields have been initialized.
- Steps to create an object are:
  1. Allocate the memory for the fields;
  2. Initialize fields;
  3. Invoke the constructor;
  4. Return the reference.

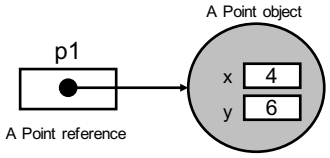
```
class Demo1{
    int di = 5;        // 1.
    int dj;            // 2. dj = 0

    Demo1() {
        dj = 4;        // 3-1.
    }

    Demo1(int j) {
        dj = j;        // 3-2.
    }
}
```

(always and first; only one)

```
Demo1 oa1 = new Demo1();    // case 3-1.
Demo1 oa2 = new Demo1(6);   // case 3-2.
oa2.di = 948;
```


Initializing an Object

- The Point class has only 1 constructor.
- The constructor takes 2 integer arguments.

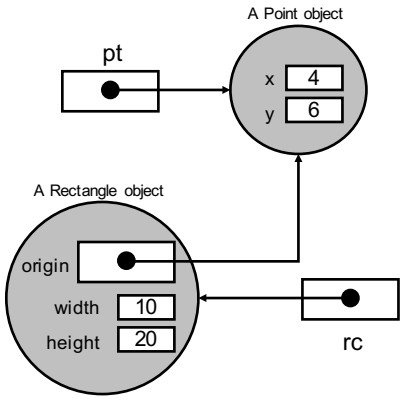



```
public class Point {
    public int x;        // = 0
    public int y;        // = 0

    public Point(int a,int b) {
        x = a;
        y = b;
    }
}

Point p1 = new Point(4,6);
```

Example: Initializing an Object

```
public class Rectangle {
    private Point origin;
    private int width = 0;
    private int height = 0;
    // no argument constructor
    public Rectangle(){
        origin = new Point(0,0);
    }
    // constructor with 3 parameters
    public Rectangle(Point p, int w,
    int h) {
        origin = p;
        width = w;
        height = h;
    }
    public int getArea(){
        return width * height;
    }
}
```

```
Point pt = new Point(4,6);
Rectangle rc = new Rectangle(pt,10,20);
```



Defining Methods

- A typical method declaration:

```
public double methodName(double parameter, …){
    …        // method body
}
```

- A method declaration has 6 components:
  - A **modifier** – such as public, private, etc.
  - The **return type** – the data type of the value returned by the method, or void if no value is returned.
  - The **method name** - names should begin with a letter (a-z, A-Z), followed by letters, digits, dollar signs, or underscores.
  - The **parameter list** in parenthesis – a list delimited by commas. A method may have no parameters.
  - An **exception list** – exceptions are used in error handling.
  - The **method body** – the method's code.

Invoking an Object's Methods

- An object's reference is used to invoke a method on that object, as in:
  objectReference.methodName(argumentList);
- For example (see the previous slide):
  - int area = new Rectangle().getArea();
- A reference to an object must be initialized before being used.
- Examples:

```
Rectangle rec;                  // not initialized
int area = rec.getArea();       // error

rec = new Rectangle();          // initialization
area = rec.getArea();           // correct
```


Parameters of a Method or Constructor

- Parameters refer to the **list of variables** in a method declaration.
- Arguments are the **actual values** that are passed in when the method is invoked.
- The declaration for a method or a constructor declares the **parameters** for that method or constructor:
  - public void method(int a,int b,char c) { … }
- The arguments must match the parameters in type and order:
  - method(4,6,'8');
  - method(4,6,8); // error

Parameter Types

- Any data type may be used for a parameter of a method or a constructor.
- For example:
  - void method(int i) { … }
  - void method(int i,String s) { … }
  - void method(String s) { … }
  - void method(String s,Object o,int i) { … }

Passing Primitive Data Type Arguments

- Primitive arguments, such as an int or a float, are passed into methods *by value.*
- If a parameter changes its value in the method, that changed value exists only within the scope of that method. When the method returns, the parameters are gone and any changes to them are lost.

```
public class PassPrimitiveByValue {
   public static void main(String[] args){
      int x = 3;
          // invoke passMethod() with x as
          // argument
      passMethod(x);
          // print x to see if its value has
          // changed
      System.out.println("After invoking
                          passMethod, x = " + x);
   }
   // change parameter in passMethod()
   public static void passMethod(int p) {
      p = 10;
   }
}
```

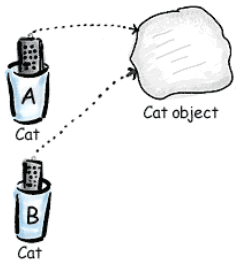
The output of this program is:

After invoking passMethod, x = 3

Passing Reference Data Type Arguments

- Java passes everything by value.
- With primitives, you get a copy of the contents.
- When you pass an object reference into a method, you are passing a COPY of the REFERENCE.
  - Example on the right: There is still just ONE Cat object. But now TWO remote controls (references) can access that same Cat object.
- You can change the Cat, using your new B reference (copied directly from A), but you can't change A.

```
Cat A = new Cat();
doStuff(A);

void doStuff(Cat B){
        // use B in some way
}
```

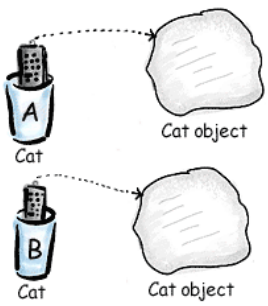


JavaWorkshop.FCSIT@buk

Passing Reference Data Type Arguments

- Example on the right: B = new Cat();
- Statement above simply "points" B to control a different object.

```
Cat A = new Cat();
doStuff(A);

void doStuff(Cat B){
  B = new Cat(); // did NOT affect
                 // the A reference
}
```



JavaWorkshop.FCSIT@buk

The Garbage Collector

- Java uses a garbage collector to free the memory used by objects that are not referenced any more.
- An object can be garbage collected when there are no references to that object.
- Garbage collection is automatic.

```
Point p1 = new Point(0,0);
Point p2 = p1;
```

- After execution of the two statements above, there are 2 references to the Point(0,0) object.

```
p1 = null;    // 1 reference
p2 = null;    // 0 references
```

- The Point(0,0) object is eligible for garbage collection.

The *this* Keyword

- '***this***' is a reference to the current object.
- It can be used within an instance method or a constructor of a class.
- Using ***this*** with a field is the most common reason for using the 'this' keyword: Because a field is shadowed by a method or constructor parameter.

```
public class Point {
   public int x = 0;
   public int y = 0;

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

- Example above:
  - Each argument to the constructor shadows one of the object's fields—inside the constructor **x** is a local copy of the constructor's first argument. To refer to the Point field **x**, the constructor must use **this.x**.

The *this* Keyword

- 'this' can be used within a constructor to invoke another constructor of the same class.

```
class AA {
    int x,y;

    AA() {
        this(0,0);
    };

    AA(int xx,int yy) {
        x = xx;
        y = yy;
    }

}
```

Controlling Access to Members of a Class

- Access level **modifiers** determine whether other classes can access a field or invoke a method.
- There are 2 levels of access control:
  - At the top level – package-private (no explicit modifier stated) or public.
  - At the member level – package-private (no explicit modifier stated), public, private, or protected.
- When a class is declared public, it is visible to all classes everywhere.
- If a class has no **modifier**, it is visible only within its package.

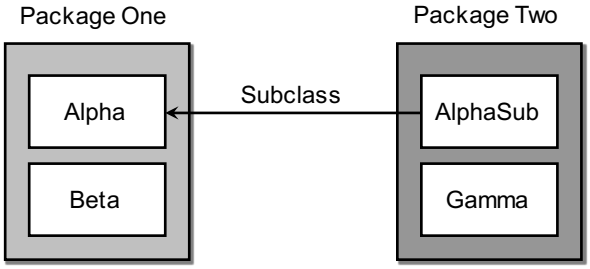
Controlling Access to Members of a Class

- The first data column indicates whether the class itself has access to the member defined by the access level.
- The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member.
- The fourth column indicates whether all classes have access to the member.

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

JavaWorkshop.FCSIT@buk

Visibility



Modifier	Alpha	Beta	AlphaSub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

This table shows where members of the alpha class are visible for each of the access modifiers that can be applied to them.

JavaWorkshop.FCSIT@buk

Tips on Choosing an Access Level

- Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
- Avoid public fields except for constants. (Many of the examples in the book use public fields. This may help to illustrate some points concisely, but is not recommended for **production code**.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Example: Access Level

```
public class AA {
    private int aak = 3;
    public float aaf = 0.2F;

    public void methodA() {
       aak = 4;
    }

}
```

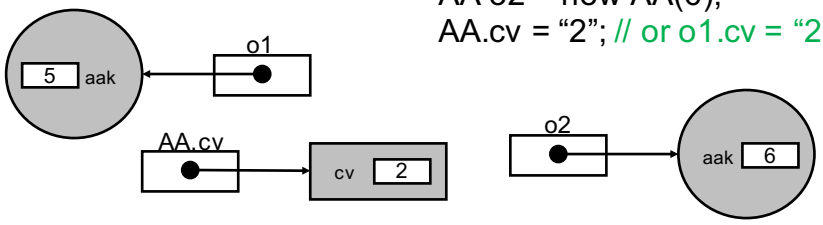
```
public class BB {
        private void method() {
                AA oa = new AA();

                oa.aak = 5;   // error
                oa.aaf = 5.0F;
        }

}
```

- Class BB may access a public field or method of class AA.
- Class BB may not access a private field or method of class AA.

Class Variables (or Static Fields)

- Each object has its own copy of instance variables (fields).
- Class variables (*static* fields) are common to all objects of that class (only one copy exist).
- Class variables are associated with the class.

```
class AA {
    int aak;
    static String cv;

    AA(int i) { aak = i; }
}

AA o1 = new AA(5);
AA o2 = new AA(6);
AA.cv = "2"; // or o1.cv = "2";
```



Class Methods (or Static Methods)

- Static methods, like static fields, belong to the class.
- A class method can be invoked with the class name or an object reference:
  - className.method(args)

    AA.clsMethod();

  - instanceName.method(args)

    AA oa = new AA();
    oa.clsMethod();
- A common use for static methods is to access static fields.

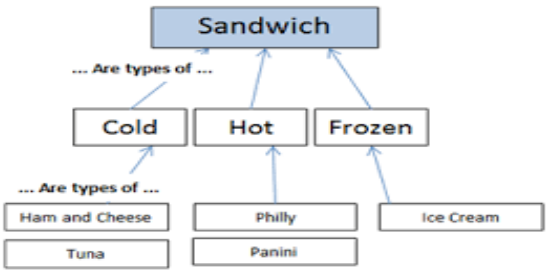
```
class AA {
    int aak;
    static int cv;

    void method(){
        this.aak = 3;
        cv = 2;
    }
    static void clsMethod(){
        cv = 4;
        this.aak = 5;   // error
    }
}
```

Constants

- A constant uses the *final* keyword.
- A constant can be initialized, but **cannot** change its value.
- Examples:
  - final int value = 3; // initialized
    value = 5; // error
  - final int value; // not initialized
    value = 5; // initialized
    value = 6; // error

Summary of Classes and Objects

- A class declaration names the class and encloses the class body between brackets.
- A class uses fields (they contain state information) and methods (they implement class behavior).
- Constructors initialize a new instance of a class. They use the name of the class and look like methods without a return type.
- You create an object from a class by using the ***new*** operator and constructor.
- The garbage collector automatically cleans up unused objects.
  - An object is unused if there is no references to it.



Inheritance part I

Contents

- Inheritance: Key Definitions
- The Purpose of Inheritance
- An Example of Inheritance

Inheritance: Key Definitions

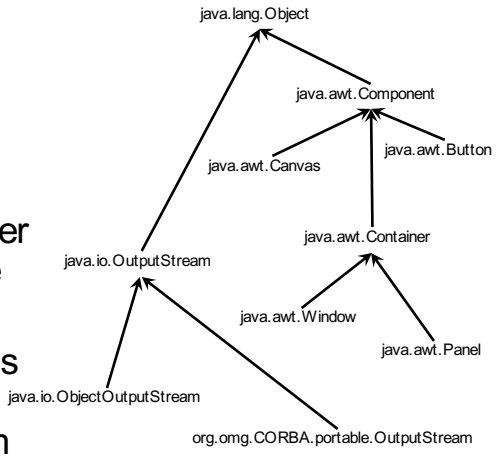
- A class that is derived from another class is called a ***subclass*** (also a ***derived class***, ***extended class***, or ***child class***).
- The class from which the subclass is derived is called a ***superclass*** (also a ***base class*** or a ***parent class***).
- Every class has one and only one direct superclass (single inheritance).
- Only *java.lang.Object* has no superclass.
- Every class is implicitly a subclass of Object.

The Purpose of Inheritance

- The idea of inheritance is simple but powerful:
  - When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class.
  - In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

Single Inheritance

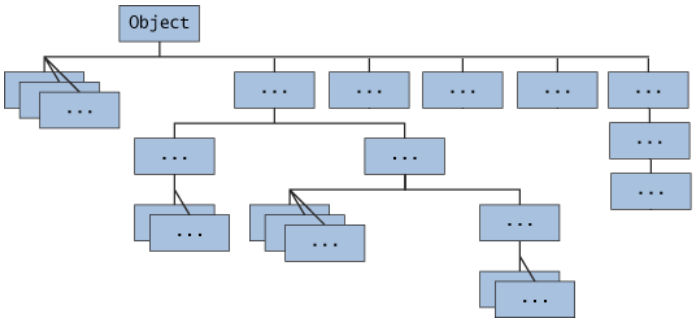
- In Java, a class can have only 1 direct **superclass** (*single inheritance*)
- A class that is the **superclass** to another class, may also have a **superclass**
- A hierarchy of classes can be formed, and they all *descend* from all the **superclasses**



JavaWorkshop.FCSIT@buk

The Java Platform Class Hierarchy

- The Object class, defined in the java.lang package, defines and implements behavior common to all classes—including the ones that you write.
- In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



JavaWorkshop.FCSIT@buk

Inheritance: A Closer Look

- A subclass inherits all the *public* and *protected members* (fields, methods) from its superclass.
  - When a subclass inherits a member, it is as if the subclass defined the member itself.
- Constructors are not members, so they are not inherited by subclasses.
- The constructor of the superclass can be invoked from the subclass by using the keyword *super*.

JavaWorkshop.FCSIT@buk

A Hierarchy of Bicycle Classes

Bicycle

Bicycle (superclass): more general features

RoadBike MountainBike TandemBike

RoadBike, MountainBike, TandemBike (subclasses): more specific features

JavaWorkshop.FCSIT@buk

Comments on the Previous Slide

- In chapter 2, you considered an example of the bicycle class.
- Different kinds of objects often have a certain amount in common with each other.
  - Bicycle is superclass (more general features).
  - Mountain bikes, road bikes, and tandem bikes share the characteristics of bicycles:
    - —current speed,
    - —current pedal cadence,
    - —current gear.
  - MountainBike, RoadBike, and TandemBike are subclasses of Bicycle (more specific features).

An Example of Inheritance

```
public class Bicycle {        // the Bicycle class has three fields
    protected int cadence;
    protected int gear;
    protected int speed;        // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }                                // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }
    public void setGear(int newValue) {
        gear = newValue;
    }
    public void applyBrake(int decrement) {
        speed -= decrement;
    }
    public void speedUp(int increment) {
        speed += increment;
    }
}
```

An Example of Inheritance

- A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {
                    // the MountainBike subclass adds one field
    protected int seatHeight;
                    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
                    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

- MountainBike inherits all the fields and methods of Bicycle and adds the field ***seatHeight*** and a method to set it. a new MountainBike class has four fields and five methods.

Lab: **Your Help to School Students**

- The Ministry of Education asks you to design the special program for school students. It aim is to help them understand solid geometry. The specification of the program is as follows.
- You should design and implement a *Sphere* class. It contains instant data that represents the sphere's radius (cm)(its type should be *float*). You should define the *Sphere constructor* to accept and initialize the radius. You should include a method that calculates and returns the volume of the sphere.
- Two spheres may contact each other. What is the height of their contacting point? You should create a special method that calculates this value. The method signature is as follows:

  float heightContactPoint(Sphere secondSphere)

Lab: **Your Help to School Students**

- You should create a *TestSphere* class, whose main method instantiates three spheres. Two spheres should have the same radius. The method should print the height of the contacting point for each pair of spheres and the volume of each sphere.
- You may look at the implementation of the *Bicycle* class (see lecture slides.)

END