Introduction to Java

Overview of Part 1 of the Course

- Demystifying Java: Simple Code
- Introduction to Java
- An Example of OOP in practice
- Object Oriented Programming Concepts
- OOP Concepts – Advanced
- Hints and for Java
- I/O (Streams) in Java
- Graphical User Interface Coding in Java
- Exceptions and Exception handling

Overview

- Objects and Classes
- Fields, Methods, and Constructors
- Method overloading and overridable
- Instance and class variables
- Object and class methods
- Visibility / Access
- Inheritance and Polymorphism
- Subclass and superclasses
- Constructor chaining
- Object reference casting
- Shadowed variables and overriding method
- Dynamic method binding
- Abstract method and interface
- Inheritance hierarchy, subclasses, and multiple inheritance
- Visibility / Access
What is an object? An object is a model of some specific thing - physical or conceptual. Man is not an object; Adam is an object. Man is an object of the class Man. There are different kinds of objects.

What then is a class? A class is the mechanism to define a collection of objects - objects of the same kind. Man is a class; Adam is an object of the class.

An object is a specific instance of a class.

Java classes and objects

In Java, the class construct conceptually gathers together the data fields and methods to define the state and behavior of each object (instance of the class). The fields and the methods are collectively referred to as the members of the class.
**Fields and Methods**

```java
class Stack {
    // fields
    int[] root = new int[100];
    int top = 0;
    // methods
    void push(int x) { root[top++] = x; }
    int pop() { return root[--top]; }
    boolean isEmpty() { return top == 0; }
}
```

- Creating a Stack object:
  - Given a Stack object, we can push an integer onto it and pop the integer off the top of the stack. We can verify that the stack is empty using the methods.
- To create a Stack object, we make an instance of the class using the `new` operator on the class name:
  ```java
  Stack s = new Stack();
  ```

**The new operator call**

- The `new` operator results in a call to the constructor method. A Stack constructor is a method to initialize a Stack object newly created by the `new` operator call.
- A constructor is a method named `ClassName` and has the same name as the class. It is a special method used to initialize the object.
- If we need to create a stack of a specific size, we can provide a constructor:
  ```java
class Stack {
    int[] root;
    // constructor
    Stack(int size) { root = new int[size]; }
}
```

**Constructors**

- However, when we provide a constructor, there will be no default constructor. We will have to say:
  ```java
  Stack s = new Stack();
  ```
- But have to say:
  ```java
  Stack s = new Stack(100); // stack of size 100
  Stack s2 = new Stack(1000); // bigger stack
  ```
  - We may need to provide another constructor to satisfy the stack size when we create the stack:
  ```java
class Stack {
    int[] root;
    // constructor
    Stack(int size) { root = new int[size]; }
    // another constructor
    Stack() { root = new int[100]; }
}
```
Constructors

- Stack class with more than one constructor...

```java
class Stack {
    int[] root;
    int top = 0;

    // constructors...
    Stack(int size) { root = new int[size]; }
    Stack() { root = new int[100]; }

    // methods...
    void push(int x) { root[top++] = x; }
    int pop() { return root[--top]; }
    boolean isEmpty() { return top == 0; }
}
```

- Different constructors must have different sequence of parameter types. (This allows the compiler to know which one to call, when we have more than one constructor.)

One constructor calling another...

- One constructor can also call another constructor.

```java
class Stack {
    int[] root;
    int top = 0;

    // constructors...
    Stack(int size) { root = new int[size]; }
    Stack() { this(100); } // Set default size to 100

    // methods...
    void push(int x) { root[top++] = x; }
    int pop() { return root[--top]; }
    boolean isEmpty() { return top == 0; }
}
```

- Note the syntax of how one constructor calls another.

Method overloading

- When the same method name is used to refer to more than one method, the name is *overloaded*.

```java
class Stack {
    int sum(int from, int to) { ... }
    int sum(int toLevel) { ... }
    int sum() { ... } // sum up the whole stack.
}
```

- Note that the signature of a method does not include the method’s return type.
Garbage collection and finalize

• Java applies a technique called garbage collection to relieve the programmer from the task of managing memory space no longer in use. While Java acquires memory space to construct a new object upon each operator call, with garbage collection, Java does NOT reclaim the memory space not in use until it is deemed necessary.

• A class may optionally define a finalize method, which is called when the system reclaims the space in garbage collection. The method must be named finalize, and has no parameters and no return type. There is no guarantee whether or not the finalize method will be called, nor when it will be called.

Instance variable & class variable

• The members of a class consist of fields and methods. Each field defines an instance variable of each object of the class, except when the field is declared to be static.

class Circle {
    double x,y,r;  // (x,y) coordinates and radius
    static int circle_count;  // number of circles created
    ...
}

• Every object of the class has its own x,y coordinates and its own radius. Each field - x, y, r - identifies an instance variable of a Circle object.

• But if we want to keep a count of how many objects have been created in the class Circle...

• We need to declare a static field.

• The static field circle_count is a class variable, because there is only one copy of it for the whole class. It models an attribute of the class, not that of each object of the class.

class Circle {
    double x,y,r;  // (x,y) coordinates and radius
    static int circle_count;  // number of circles created
    ...
}

• If each of the Circle constructors would increment circle_count, it would keep a count of how many Circle objects have been created, provided that we properly initialize it to zero.
Class variable

- Class Circle:
  - double x, y;  // (x,y) coordinates
  - int circle_count;  // number of circles created
  - Circle(double x, double y, double r) { this.x = x; this.y = y; circle_count++; }

- To get the number of Circle objects created:
  - System.out.println(Circle.circle_count);

- Note that we refer to a static member field without a reference to an object.
- Java also provides a special method for the initialization of static member fields.

Static initializer

- The static initializer is a special method. The method must be declared static, and it has no name, no parameters, and no return type.

```java
class Circle {
  double x, y;  // (x,y) coordinates
  static int circle_count;  // number of circles created
  Circle(double x, double y, double r) { … }
  static { circle_count = 0; }  // static initializer
}
```
- The static initializer method is invoked automatically, invoked once and once only when the class is loaded for the first time.

Object method & class method

- The members of a class consist of methods. Each method defines an object method unless it is declared static. In that case, the method is called a class method.

```java
class Circle {
  double x, y, r;  // (x,y) coordinates and radius
  static final double PI = 3.1416;
  // Object method to compute area of a circle.
  double area() { return PI * r * r; }
  // Class method to compare which of two circles is bigger.
  static Circle bigger(Circle c1, Circle c2) {
    if (c1.r > c2.r) return c1;
    else return c2;
  }
}
```
- `area()` is an object method while `bigger(c1,c2)` is declared static, and is therefore a class method.
Object method & class method

- Note the method calls:
  ```java
  Circle c1 = new Circle(0.0, 0.0, 3.0);
  double a1 = c1.area();  // area of circle c1.
  Circle c2 = new Circle(1.0, 1.0, 2.5);
  double a2 = c2.area();  // area of circle c2.
  Circle c3 = Circle.bigger(c1, c2);
  double a3 = c3.area();  // area of c3, the bigger circle.
  ```
- To invoke an object method, we must do so with a reference to an object of the class. But we can invoke a class method simply using the class name.
- In the method body which defines a class method there is no this reference to refer to an object of the class.

Access / Visibility

- In Java, we can apply access/visibility modifiers to specify the visibility of (or access to) the names declared for use. Other parts of the program can use the names only when the name is visible there (i.e., access to the name is permitted).
- These modifiers apply to both objects as well as classes.

Visibility / Access

- Classes are partitioned into packages, and packages are organized hierarchically into a directory tree.
- Two visibility settings apply to classes:
  - default – no modifier – visible to the package only
  - public – visible everywhere
- Four visibility settings apply to class members:
  - default – no modifier – visible to the package only
  - public – visible everywhere
  - private – visible only within class scope
  - protected – visible to the package and subclasses
- Java source file must have the name extension .java. Each file allows only one public class, named by the same name as the file.
package x;

public class Circle {
    private double x, y, r; // (x, y) coordinates and radius
    protected int num = Circle.count = 0;
    public static final double PI = 3.1416;
    public Circle(int x, int y, int r) {
    }  // OK – constructor.
    public double area() { return PI * r * r; }
    public static Circle bigger(Circle c1, Circle c2) {
        if (c1.r > c2.r) return c1; else return c2;
    }
    ...  // End of class scope
}

class T {
    static void main(String[] args) {
        Circle c1; // OK – Circle is public
        c1 = new Circle(0.0, 0.0, 2.5); // OK – public ctor.
        double radius = c1.r; // ERROR: private, no access.
        double r_square = c1.area() / Circle.PI; // OK.
        int num_circles = Circle.circle_count; // ERROR.
    }
}

---

### Visibility / Access

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Class Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>(No modifier)</td>
<td>Visible in package</td>
<td>Visible in package.</td>
</tr>
<tr>
<td>private</td>
<td>Not accessible.</td>
<td>Visible in package and subclass.</td>
</tr>
<tr>
<td>protected</td>
<td>Not accessible.</td>
<td>Visible in package and subclasses.</td>
</tr>
</tbody>
</table>

---

### Inheritance and Polymorphism
Inheritance and Polymorphism

• Subclass and super class
• Constructor chaining
• Object reference casting
• Shadowed variable and overridden method
• Dynamic method binding
• Abstract method and abstract class
• Inheritance hierarchy
• Interfaces
• Multiple inheritance
• Visibility / Access

Subclass and Super class

• A class is a construct to define a collection of objects of the same kind. A class construct may be derived to define a subset of that collection. We call it a subclass of the super class.

    class Circle {
        double x, y, r;
        double area() { … } …
    }

    class GraphicCircle extends Circle {
        Color outline, inside;
        …
    }

• Every object of the subclass GraphicCircle can also be treated as an object of the super class Circle.

Subclass and Super class

• There may be differences in response and behavior; every object of the subclass GraphicCircle can also be treated as an object of the super class Circle, but NOT the other way round.

    GraphicCircle gc = new GraphicCircle(…);
    Color c_outside = gc.outline;
    Color c_inside = gc.inside;
    double center_x = gc.x;
    double center_y = gc.y;
    double radius = gc.r;
    double area = gc.area();  // OK also for method area().
Subclass and Superclass

- Since we can refer to members of the super class as if they are members of the subclass, we say that the subclass inherits from the super class.
- Therefore, the construction of a GraphicCircle object of the subclass must also imply the construction of a Circle object (of the super class). In Java, the construction of a Circle object implicitly calls the constructor of the super class if not done explicitly.
- Note in the following example, the places where an implicit call to a super class constructor may be inserted.
- This is known as Constructor Chaining.

```
constructor chaining

class Circle {
  Circle(double x, double y, double r){
    this.x = x; this.y = y; this.r = r;
  }
  Circle() {
    this(0.0,0.0,1.0);
  }
}
class GraphicCircle extends Circle {
  GraphicCircle(double x, double y, double r, Color outline, Color inside) {
    super(x,y,r); // Explicit ctor call to super class.
    this.outline = outline;
    this.inside = inside;
  }
  GraphicCircle(Color outline) {
    this.outline = outline; // Note - inserted:
    super();
    this.inside = WHITE;
  }
  GraphicCircle() {
    this(BLACK,WHITE); // Nothing inserted.
  }
}
```

Constructor chaining -- finalizer?

- The construction of a subclass object must imply the construction of a super class object -- therefore the constructor calls are chained from subclass to super class.
- The same style of chaining does NOT apply to the finalize calls. (The finalize() method is called after the object is garbage collected -- in the finalizer.)
- Vin the following example, an implicit call to the finalize() method of the super class is not done.
- Java does not guarantee the calling of the finalize methods, nor the order in which they may follow.

```
constructor chaining -- finalizer?

```
**object casting**

- When `class B extends class A`, we can refer to an object of `class B` in terms of the `class A` object. This can happen by explicit casting, or implicit conversion in assignment, argument passing, or method return.
  
  ```java
  B b;
  A a;
  (A)b.i     // Explicit casting
  A a = b;    // Implicit conversion in assignment
  ```

- It is OK because we can treat an object of a subclass as if it is an object of the super class.

- We can treat an object of a subclass as if an object of the super class; but NOT the other way around.

**Shadowed variable**

- The subclass inherits the fields from the super class, but can also define its own fields. If the subclass defines a field using the same name as an inherited field, the field from the subclass becomes a **shadowed variable** in the subclass — it becomes inaccessible unless requested explicitly.

  ```java
  class A { int i = 1; }
  class B extends A {
    int i = 2;
    int f() { return i; } // my own i in B: 2.
    int g() { return super.i; } // i from A: 1.
  }
  ```

- B b = new B();
- System.out.println(b.f()); // prints out 2.
- System.out.println(b.g()); // prints out 1.

**Overriding Method**

- The subclass inherits the methods from the super class, but can also define its own methods. If the subclass defines a method with the **same signature** as an inherited method, the method is called an **overriding method**, because the inherited method becomes inaccessible unless explicitly requested.

  ```java
  class A {
    int f() { return 1; }
  }
  ```

- class B extends A {
  ```java
    int f() { return 2; }
    int g() { return super.f(); } // prints out 1.
  ```
- B b = new B();
- System.out.println(b.f()); // prints out 2.
- System.out.println(b.g()); // prints out 1.

  ```java
  ```
### Shadowed Variable & Overriding Method

```java
Class A {
    int i = 1;
    int f() { return i; }
}

Class B extends A {
    int i = 2;
    int f() { return -i; }
}
```

```java
B b = new B();
System.out.println(b.i);  // 2
System.out.println(b.f()); // -2
A a = b;  // treat it an object of class A.
System.out.println(a.i);  // 1
System.out.println(a.f()); // -2
```

- Since class B extends class A, we can treat an object of class B as if it is an object of class A.
  ```java
  A a = b; // Treat object b as if it is of class A.
  ```

- When treated as an object of class A, the name of a shadowed variable will refer to that of class A. Hence, `a.i` gives 1. But the name of an overriding method now refers to the method defined for the class the object actually is, i.e., class B. Hence, `a.f()` gives -2.

- **Polymorphism** is the phenomenon in which treating similar objects in the same way leads to different responding behavior according to the difference in objects.

Java supports polymorphism by **dynamic method binding**

### Dynamic method binding

```java
class Shape {
    double s;
    double area() { return -1.0; }
    Shape(final double s) { this.s = s; }
}

class Circle extends Shape {
    final double PI = 3.1416;
    double area() { return PI*s*s; }
    Circle(final double r) { super(r); }
}

class Square extends Shape {
    double area() { return s*s; }
    Square(final double s) { super(s); }
}

class Rectangle extends Square {
    double oside;
    double area() { return s*oside; }
    Rectangle(final double s, final double oside) { super(s); this.oside = oside; }
}
```
Dynamic method binding

```java
class Shape {
    abstract double area();
    Shape(final double s) { this.s = s; }
}
class Circle extends Shape {
    double s;
    Circle(final double s) { super(s); }
    double area() { return Math.PI * s * s; }
}
class Square extends Shape {
    double s;
    Square(final double s) { super(s); }
    double area() { return s * s; }
}
class Rectangle extends Square {
    double l, w;
    Rectangle(final double l, w) { super(l, w); }
    double area() { return l * w; }
}
Shape c = new Circle(3.0);
Shape s = new Square(5.0);
Shape r = new Rectangle(4.0,6.0);
double ca = c.area(); // 28.3 of Circle
double sa = s.area(); // 25.0 of Square
double ra = r.area(); // 24.0 of Rectangle
```

Observe that the correct area() method was called.

Abstract method

- Note that the area() method of class Shape is not used at all. In fact, the method is meaningless for the class. We need the area() method only because we want to handle uniform way to treat their subclasses.
- In other words, area of a shape makes sense only when it is a "real" shape.
- Java allows us to declare a method but supply no method body to define its behavior - it is therefore called an abstract method.
- abstract class Shape {
    double s;
    abstract double area();
    Shape(final double s) { this.s = s; }
}

Abstract class

- A class with at least one abstract method is called an abstract class. (The modifier abstract is optional.)
- There can be NO instances of an abstract class.
- A subclass can implement and override an abstract method.
- The subclass is not abstract (concrete) when all abstract methods have an overriding method implemented. We can then have objects (instances) of the subclass and treat them as if they were objects of the abstract class.
- NOTE therefore in the inheritance hierarchy: the super class of an abstract class is always an abstract class; the subclass of a non-abstract superclass is always a non-abstract (concrete) class.
Inheritance hierarchy

- In Java, each class can have at most one and only one super class. That is, a class cannot inherit from more than one class.
- In Java, there is a class Object - each class is automatically a descendant of the class Object, that is, each class is either a subclass of Object, or that it is a descendant of some subclass of Object.
- Hence, in Java, the inheritance hierarchy shows a tree rooted at the class Object.

Interface

- An abstract class represents the way we can treat the objects of its subclasses. The form of treatment - as represented in the collection of methods - is partially implemented.
- An interface is similar to an abstract class:
  ```java
  interface Drawable {
    static final double PI = 3.1416;
    void setColor(Color c);
    void setPosition(double x, double y);
    void draw(Window w);
  }
  ```
- All the methods of the interface must be abstract, furthermore, all the fields must be static final.

Multiple Inheritance and Interfaces

- Many OOP use multiple inheritance to give objects characteristics of multiple parents.
- Useful, but potentially a landmine if both parents have a method of the same name. For these reasons, Java does not allow multiple inheritance.
- Java uses interfaces as an alternative to avoid these problems by allowing the equivalent of multiple inheritance.
- Interfaces are implemented frequently as a means for providing class hierarchies.
Interface

- An **interface** represents a way to treat objects.
- When a class implements overriding methods for all the abstract methods in an interface, we say the class **implements** the interface.

```java
class Circle {
    double x, y, r;
    static final double PI = 3.1416;
    double area() { return PI * r * r; }
}
class DrawableCircle extends Circle implements Drawable {
    Color c;
    void setColor(Color c) { this.c = c; }
    void setPosition(double x, double y) {
        this.x = x; this.y = y;
    }
    void draw(Window w) {
        …
    }
}
```

- When a class implements an interface, we can treat the objects of the class according to the treatment represented in the interface – apply the methods of the interface on the object.
- Hence, we can have object reference for an interface referring to objects of the class implementing the interface.
- Class DrawableCircle implements interface Drawable …
  Drawable dx = new DrawableCircle(…);
  dx.setColor(YELLOW);
  dx.setPosition(10.0, 10.0);
  dx.draw(Desktop_Window);

Interface

- Much like subclass extending an abstract class, we can extend an interface into another interface (sometimes called subinterface) – as long as we continue to observe the rules of abstract methods and constant fields. The extended interface is still an interface.

```java
interface Drawable {
    void setColor(Color c);
    void setPosition(double x, double y);
    void draw(Window w);
}
interface Fillable extends Drawable {
    void setImage(Image);
    void setRegion(Polygon);
    void fill(Window w);
}
```
Multiple interfaces

• An important result of the restriction in interfaces (compared to abstract classes) is that an interface can unambiguously extend multiple interfaces.

```java
interface Draw_and_HighLight {
    // ... methods ...
}
```

• A class can also implement multiple interfaces.

```java
class LightedButton extends Shape implements Drawable, Highlight {
    // ... methods ...
}
```

Visibility / Access

• The same visibility/access modifiers apply...

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class / Interface</th>
<th>Class Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Visible everywhere.</td>
<td>Visible everywhere.</td>
</tr>
<tr>
<td>private</td>
<td>Not applicable.</td>
<td>Visible only within class scope.</td>
</tr>
<tr>
<td>protected</td>
<td>Not applicable.</td>
<td>Visible in package and subclasses.</td>
</tr>
</tbody>
</table>