Inheritance

Department of Data Science
& Engineering, DCA, MIT

Inheritance

- Same inheritance concept of C++ in Java with some modifications
  - One class inherits the other using ***extends*** keyword
  - The classes involved in inheritance are known as ***superclass*** and ***subclass***
  - ***Multilevel*** inheritance but no ***multiple*** inheritance
  - There is a special way to call the superclass's ***constructor***
  - There is automatic ***dynamic method dispatch***
- Inheritance provides code reusability (code of any class can be used by extending that class)

Simple Inheritance

```java
class A {
    int i, j;

    void showij() {
        System.out.println(i+" "+j);
    }
}

class B extends A{
    int k;

    void showk() {
        System.out.println(k);
    }

    void sum() {
        System.out.println(i+j+k);
    }
}
```

```java
public class SimpleInheritance {
    public static void main(String[] args) {
        A superOb = new A();
        superOb.i = 10;
        superOb.j = 20;
        superOb.showij();
        B subOb = new B();
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        subOb.showij();
        subOb.showk();
        subOb.sum();
    }
}
```


P : SimpleInheritance.java

Inheritance and Member Access

```java
class M {
    int i;
    private int j;

    void set(int x, int y) {
        i = x;
        j = y;
    }
}

class N extends M {
    int total;

    void sum() {
        total = i + j;
        // Error, j is not accessible here
    }
}
```

```java
public class SimpleInheritance2 {
    public static void main(String[] args) {
        N obj = new N();
        obj.set(10, 20);
        obj.sum();
        System.out.println(obj.total);
    }
}
```

- A class member that has been declared as private will remain private to its class
- It is not accessible by any code outside its class, including subclasses

P : SimpleInheritance2.java

Practical Example

```
3  class Box {
4      double width, height, depth;
5
6      Box(Box ob) {
7          width = ob.width; height = ob.height; depth = ob.depth;
8      }
9
10     Box(double w, double h, double d) {
11         width = w; height = h; depth = d;
12     }
13
14     Box() { width = height = depth = 1; }
17
18     Box(double len) { width = height = depth = len; }
21
22     double volume() { return width * height * depth; }
25 }
26
27 class BoxWeight extends Box {
28     double weight;
29
30     BoxWeight(double w, double h, double d, double m) {
31         width = w; height = h; depth = d; weight = m;
32     }
33 }
```


P : RealInheritance.java

Superclass variable reference to Subclass object

```
34
35  public class RealInheritance {
36      public static void main(String[] args) {
37          BoxWeight weightBox = new BoxWeight( w: 3,  h: 5,  d: 7,  m: 8.37);
38          System.out.println(weightBox.weight);
39          Box plainBox = weightBox; // assign BoxWeight reference to Box reference
40          System.out.println(plainBox.volume()); // OK, volume() defined in Box
41          System.out.println(plainBox.weight); // Error, weight not defined in Box
42          Box box = new Box( w: 1,  h: 2,  d: 3); // OK
43          BoxWeight wbox = box; // Error, can't assign Box reference to BoxWeight
44      }
45  }
46
```


P : RealInheritance.java

Using super to call Superclass Constructors

```
3   class BoxWeightNew extends Box {
4       double weight;
5
6       BoxWeightNew(BoxWeightNew ob) {
7           super(ob);
8           weight = ob.weight;
9       }
10
11      BoxWeightNew(double w, double h, double d, double m) {
12          super(w, h, d);
13          weight = m;
14      }
15
16      BoxWeightNew() {
17          super(); // must be the 1st statement in constructor
18          weight = 1;
19      }
20
21      BoxWeightNew(double len, double m) {
22          super(len);
23          weight = m;
24      }
25
26      void print() {
27          System.out.println("Box(" + width + ", " + height +
28                             ", " + depth + ", " + weight + ")");
29      }
30  }
```

super( ) must always be the first statement executed inside a subclass' constructor

P : SuperTest.java

Using super to call Superclass Constructors

```
31
32  public class SuperTest {
33      public static void main(String[] args) {
34          BoxWeightNew box1 = new BoxWeightNew(10, 20, 15, 34.3);
35          BoxWeightNew box2 = new BoxWeightNew(2, 3, 4, 0.076);
36          BoxWeightNew box3 = new BoxWeightNew();
37          BoxWeightNew cube = new BoxWeightNew(3, 2);
38          BoxWeightNew clone = new BoxWeightNew(box1);
39          box1.print();
40          box2.print();
41          box3.print();
42          cube.print();
43          clone.print();
44      }
45
46  }
47
```


P : SuperTest.java

Using super to access Superclass hidden members

```
3  class C {
4      int i;
5      void show() {
6      }
7  }
8
9  class D extends C {
10     int i; // this i hides the i in C
11
12     D(int a, int b) {
13         super.i = a; // i in C
14         i = b; // i in D
15     }
16
17     void show() {
18         System.out.println("i in superclass: " + super.i);
19         System.out.println("i in subclass: " + i);
20         super.show();
21     }
22 }
23
24 public class UseSuper {
25     public static void main(String[] args) {
26         D subOb = new D( a: 1,  b: 2);
27         subOb.show();
28     }
29 }
```


P : MultilevelInheritance.java

Multilevel Inheritance

```java
class X {
    int a;
    X() {
        System.out.println("Inside X's constructor");
    }
}

class Y extends X {
    int b;
    Y() {
        System.out.println("Inside Y's constructor");
    }
}

class Z extends Y {
    int c;
    Z() {
        System.out.println("Inside Z's constructor");
    }
}

public class MultilevelInheritance {
    public static void main(String[] args) {
        Z z = new Z();
        z.a = 10;
        z.b = 20;
        z.c = 30;
    }
}
```

Inside X's constructor
Inside Y's constructor
Inside Z's constructor

P : MultilevelInheritance.java

Method Overriding

```java
class Base {
    int a;
    Base(int a) {
        this.a = a;
    }
    void show() {
        System.out.println(a);
    }
}

class Child extends Base {
    int b;

    Child(int a, int b) {
        super(a);
        this.b = b;
    }

    // the following method overrides Base class's show()
    @Override // this is an annotation (optional but recommended)
    void show() {
        System.out.println(a + ", " + b);
    }
}
```

```java
public class MethodOverride {
    public static void main(String[] args) {
        Child o = new Child( a: 10,  b: 20);
        o.show();
        Base b = o;
        b.show(); // will call show of Override
    }
}
```


P : MethodOverride.java

Dynamic Method Dispatch

```
3  class P {
4      void call() {
5          System.out.println("Inside P's call method");
6      }
7  }
8  class Q extends P {
9      void call() {
10         System.out.println("Inside Q's call method");
11     }
12 }
13 class R extends Q {
14     void call() {
15         System.out.println("Inside R's call method");
16     }
17 }
18
19 public class DynamicDispatchTest {
20     public static void main(String[] args) {
21         P p = new P(); // object of type P
22         Q q = new Q(); // object of type Q
23         R r = new R(); // object of type R
24         P  x;          // reference of type P
25         x = p;         // x refers to a P object
26         x.call();      // invoke P's call
27         x = q;         // x refers to a Q object
28         x.call();      // invoke Q's call
29         x = r;         // x refers to a R object
30         x.call();      // invoke R's call
31     }
32 }
```

- **DMD is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.**
- **DMD is a way Java implement run time polymorphism.**
- **When an overridden method is called with super class reference. Java creates different versions of an overridden method.**

P : DynamicDispatchTest.java
FindAreas.java

Abstract Class

- ***Why abstract Class?***
- ***Advantage of Abstract class.***
- ***abstract class A***
- contains abstract method ***abstract method f()***
- No instance can be created of an abstract class
- The subclass must implement the abstract method
- Otherwise the subclass will be a abstract class too

Abstract Class

```
3  abstract class S {
4      // abstract method
5      abstract void call();
6      // concrete methods are still allowed in abstract classes
7      void call2() {
8          System.out.println("This is a concrete method");
9      }
10 }
11
12 class T extends S {
13     void call() {
14         System.out.println("T's implementation of call");
15     }
16 }
17
18 class AbstractDemo {
19     public static void main(String args[]) {
20         //S s = new S(); // S is abstract; cannot be instantiated
21         T t = new T();
22         t.call();
23         t.call2();
24     }
25 }
```


P: AbstractAreas.java
AbstractDemo.java

Anonymous Subclass

```
3  abstract class S {
4      // abstract method
5      abstract void call();
6      // concrete methods are still allowed in abstract classes
7      void call2() {
8          System.out.println("This is a concrete method");
9      }
10 }
11
12 class AbstractDemo {
13     public static void main(String args[]) {
14         //S s = new S(); // S is abstract; cannot be instantiated
15         S s = new S() {
16             void call() {
17                 System.out.println("Call method of an abstract class");
18             }
19         };
20         s.call();
21     }
22 }
```

- **Make your code more concise.**
- **Enable you to declare and instantiate a class at the same time.**
- **They are like local classes except that they do not have a name.**

Using final with Inheritance

To prevent overriding

```
class A {
    final void f() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void f() { // Error! Can't override.
        System.out.println("Illegal!");
    }
}
```

To prevent inheritance

```
final class A {
    //...
}

// The following class is illegal.
class B extends A { // Error! Can't subclass A
    //...
}
```

Object Class

- There is one special class, ***Object***, defined by Java
- All other classes are subclasses of Object
- That is, Object is a superclass of all other classes
- This means that a reference variable of type Object can refer to an object of any other class
- Also, since arrays are implemented as classes, a variable of type Object can also refer to any array

Object Class Methods

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Object's toString()

- The **toString**( ) method returns a string that contains a description of the object on which it is called
- Also, this method is automatically called when an object is output using println()
- Many classes override this method
- Doing so allows them to provide a description specifically for the types of objects that they create

Object's toString()

```
3   class Point {
4       int x, y;
5
6       Point(int x, int y) {
7           this.x = x;
8           this.y = y;
9       }
10
11      @Override
12      public String toString() {
13          return "(" + x + ", " + y + ")";
14      }
15  }
16
17  public class ObjectTest {
18      public static void main(String[] args) {
19          Point p1 = new Point( x: 10, y: 20);
20          // without override toString() method the
21          // following will print something like this
22          // Point@3cd1a2f1
23          System.out.println(p1);
24      }
25  }
26
```

Object's equals() and hashCode()

- == is a reference comparison, whether both variables refer to the same object
- Object's **equals**() method does the same thing
- String class override **equals**() to check contents
- If you want two different objects of a same class to be equal then you need to override **equals**() and **hashCode**() methods
  - **hashCode**() needs to return same value to work properly as keys in Hash data structures

Object's equals() and hashCode()

```java
import java.util.HashMap;
import java.util.Objects;

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

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof Point)) {
            return false;
        }
        Point p = (Point) o;
        if (p.x == this.x && p.y == this.y) return true;
        return false;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

```java
public class ObjectTest {
    public static void main(String[] args) {
        Point p1 = new Point( x: 10, y: 20);
        Point p2 = new Point( x: 10,  y: 20);
        System.out.println(p1.equals(p2));
        System.out.println(p1 == p2);
        HashMap m = new HashMap();
        m.put(p1, "Hello");
        System.out.println(m.get(p2));
    }
}
```

Local Variable Type Inference and Inheritance

- A superclass reference can refer to a derived class object in Java
- When using local variable type inference, the inferred type of a variable is based on the declared type of its initializer
  - Therefore, if the initializer is of the superclass type, that will be the inferred type of the variable
  - It does not matter if the actual object being referred to by the initializer is an instance of a derived class

Local Variable Type Inference and Inheritance

```java
class A {
    int a;
}
class B extends A {
    int b;
}
class C extends B {
    int c;
}
public class InheritanceVarDemo {
    static A getObject(int type) {
        switch(type) {
            case 0: return new A();
            case 1: return new B();
            case 2: return new C();
            default: return null;
        }
    }
    public static void main(String[] args) {
        var x = getObject( type: 0);
        var y = getObject( type: 1);
        var z = getObject( type: 2);
        System.out.println(x.a);
        System.out.println(y.b);
                    // Error, A doesn't have b field
        System.out.println(z.c);
                    // Error, A doesn't have c field
    }
}
```

The inferred type is determined by the return type of getObject( ), not by the actual type of the object obtained. Thus, all three variables will be of type A

Static

Static Variables

- When a member (both methods and variables) is declared static, it can be accessed before any objects of its class are created, and without reference to any object
- Static variable
  - Instance variables declared as static are like global variables
  - When objects of its class are declared, no copy of a static variable is made

Static Methods & Blocks

- Static method
  - They can only call other static methods
  - They must only access static data
  - They cannot refer to ***this*** or ***super*** in any way
- Static block
  - Initialize static variables.
  - Get executed exactly once, when the class is first loaded

Static

```
3   public class StaticTest {
4       static int a = 3, b;
5       int c;
6
7       static void f1(int x) {
8           System.out.println("x = " + x);
9           System.out.println("a = " + a);
10          System.out.println("b = " + b);
11          // System.out.println("c = " + c); // Error
12      }
13      int f2() {
14          return a*b;
15      }
16      static {
17          b = a*4;
18          // c = b; // Error
19      }
20      public static void main(String[] args) {
21          f1(42); // StaticTest.f1(84);
22          System.out.println("b = " + b);
23          //System.out.println("Area = " + f2());   // Error
24      }
25  }
```

F1(42) can be accessed without object creation.

Final

- Declare a final variable, prevents its contents from being modified
- final variable must initialize when it is declared
- It is common coding convention to choose all uppercase identifiers for final variables

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Nested and Inner Classes

Nested Classes

- It is possible to define a class within another classes, such classes are known as nested classes
- The scope of nested class is bounded by the scope of its enclosing class. That means if class B is defined within class A, then B doesn't exists without A
- The nested class has access to the members (including private!) of the class in which it is nested
- The enclosing class doesn't have access to the members of the nested class

Nested Classes

```
// Demonstrate an inner class.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

inner.display() not accessible by Outer class

Output from this application is shown here:

```
display: outer_x = 100
```

Static Nested Classes

- Two types of nested classes.
  - Static
  - Non-Static
- **A static nested** class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object
- That is, **it cannot refer to members of its enclosing class directly**. Because of this restriction, static nested classes are seldom used

Static Nested Classes

```
1  class OuterStaticInner {
2      private int outer_x = 100;
3
4      void test() {
5          Inner inner = new Inner();
6          inner.display( outer: this);
7      }
8      // this is a static nested class
9      static class Inner {
10         void display(OuterStaticInner outer) {
11             System.out.println(outer.outer_x);
12         }
13     }
14 }
15
16 public class StaticNestedClassDemo {
17     public static void main(String[] args) {
18         OuterStaticInner outer = new OuterStaticInner();
19         outer.test();
20         OuterStaticInner.Inner x = new OuterStaticInner.Inner();
21         x.display(outer);
22     }
23 }
```

Inner Classes (Non-Static)

- The most important type of nested class is the inner class
- An **inner class is a non-static nested class**
- It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do
- Thus, an inner class is fully within **the scope of its enclosing class**

Inner Classes

```
1   class Outer1
2   {
3       private int outer_x = 100;
4
5       void test() {
6           Inner inner = new Inner();
7           inner.display();
8       }
9       // this is an inner class
10      class Inner {
11          void display() {
12              System.out.println(outer_x);
13          }
14      }
15  }
16
17  public class InnerClassDemo1 {
18      public static void main(String[] args) {
19          Outer1 outer = new Outer1();
20          outer.test();
21          Outer1.Inner innerObj = outer.new Inner();
22          innerObj.display();
23      }
24  }
```

Inner Classes

```
1   class Outer2
2   {
3       int outer_x = 100;
4
5       void test() {
6           Inner inner = new Inner();
7           inner.display();
8       }
9
10      class Inner {
11          int y = 10; // y is local to Inner
12          void display() { System.out.println(outer_x); }
15      }
16
17      void showy() {
18          //System.out.println(y); // error, y not known here!
19      }
20  }
21
22  public class InnerClassDemo2 {
23      public static void main(String[] args) {
24          Outer2 outer = new Outer2();
25          outer.test();
26      }
27  }
```