Inheritance

What is inheritance?

- Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in java is that you can create new classes that are built upon existing classes.
- Using inheritance, we can reuse methods and fields of a parent class, and add new methods and fields as well.
- Inheritance represents
  - the IS-A relationship
  - aka parent-child relationship
  - aka superclass-subclass relationship

Inheriting in java

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.

```
// Create a superclass.
class A {
  int i, j;

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

// Create a subclass by extending class A.
class B extends A {
  int k;

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

Accessing the subclass and super class members

```
class SimpleInheritance {
  public static void main(String args[]) {
    A superOb = new A();
    B subOb = new B();

    // The superclass may be used by itself.
    superOb.i = 10;
    superOb.j = 20;
    System.out.println("Contents of superOb: ");
    superOb.showij();
    System.out.println();

    /* The subclass has access to all public members of
       its superclass. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Contents of subOb: ");
    subOb.showij();
    subOb.showk();
    System.out.println();

    System.out.println("Sum of i, j and k in subOb:");
    subOb.sum();
  }
}
```

Inheritance contd.

- A subclass cannot access those members of the superclass that have been declared as **private**

```
class A {
  int i; // public be default
  private int j; // private to A

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

// A's j is not accessible here.
class B extends A {
  int total;

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

A Superclass Variable Referencing a Subclass Object

```
class Box {
  double width;
  double height;
  double depth;

  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }
```

```
Box(double len) {
    width = height = depth = len;
  }

  double volume() {
    return width * height * depth;
  }
}

class BoxWeight extends Box {
  double weight; // weight of box

  // constructor for BoxWeight
  BoxWeight(double w, double h, double d, double m)
{
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
}
```

A Superclass Variable Referencing a Subclass Object

```
class RefDemo {
	public static void main(String args[]) {
		BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
		Box plainbox = new Box();
		double vol;

		vol = weightbox.volume();
		System.out.println("Volume of weightbox is " + vol);
		System.out.println("Weight of weightbox is " + weightbox.weight);
		System.out.println();

		// assign BoxWeight reference to Box reference
		plainbox = weightbox;

		vol = plainbox.volume(); // OK, volume() defined in Box
		System.out.println("Volume of plainbox is " + vol);

		/* The following statement is invalid because plainbox
		does not define a weight member. */
		//  System.out.println("Weight of plainbox is " + plainbox.weight);
	}
}
```

super keyword

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.
- super has two general forms.
  - The first calls the superclass’ constructor.
  - The second is used to access a member of the superclass that has been hidden by a member of a subclass.

super to call superclass constructors

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
    double weight; // weight of box

    // initialize width, height, and depth using super()
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }
}
```

Second use of super

- The second form of super always refers to the superclass of the subclass in which it is used.
- This usage has the following general form: **super.member**
- Here, member can be either a method or an instance variable.
- This second form of super is most applicable to situations in which member names of a subclass overrides the members by the same name in the superclass.

Example

```
class A {
        int i;
}

// Create a subclass by extending class A.
class B extends A {
        int i; // this i hides the i in A

        B(int a, int b) {
                super.i = a; // i in A
                i = b; // i in B
        }

        void show() {
                System.out.println("i in superclass: " + super.i);
                System.out.println("i in subclass: " + i);
        }
}

class UseSuper {
        public static void main(String args[]) {
                B subOb = new B(1, 2);
                subOb.show();
        }
}
```

Multilevel hierarchy scenario

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same irrespective of whether or not super( ) is used.
- If super() is not used, then the default or parameterless constructor of each superclass will be executed.

Example of constructor call in class hierarchy

```
// Create a super class.
class A {
  A() {
    System.out.println("Inside A's constructor.");
  }
}

// Create a subclass by extending class A.
class B extends A {
  B() {
    System.out.println("Inside B's constructor.");
  }
}

// Create another subclass by extending B.
class C extends B {
  C() {
    System.out.println("Inside C's constructor.");
  }
}

class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to **override** the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the **subclass**.
- The version of the method defined by the superclass will be hidden.
- If a method of subclass and superclass share the same name, but type signature is different, then the two methods are simply **overloaded**

Example of method overriding

```
class A {
  int i, j;

  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```

```
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // display k -- this overrides show() in A
  void show() {
    System.out.println("k: " + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show(); // this calls show() in B
  }
}
```

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Its Java's way to implement run-time polymorphism.
- A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

Example dynamic despatch

```
class A {
   void callme() {
     System.out.println("Inside A's callme method");
  }
}

class B extends A {
  // override callme()
  void callme() {
    System.out.println("Inside B's callme method");
  }
}

class C extends A {
  // override callme()
  void callme() {
    System.out.println("Inside C's callme method");
  }
}
```

```
class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A

    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme

    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
```

abstract classes

- We can define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- Only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Java’s solution to this is the **abstract method**.
- Certain methods be overridden by subclasses by specifying the abstract type modifier.
- These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass.
- Thus, a subclass must override them—it cannot simply use the version defined in the superclass.
- To declare an abstract method, use this general form:
  - abstract type name(parameter-list)

abstract classes(2)

- Any class that contains one or more abstract methods must also be declared abstract.
- This is done by simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is an abstract class cannot be directly instantiated with the new operator.
- We cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Example

```
abstract class A {
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

Using final in inheritance

- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.
- Methods declared as final cannot be overridden
- To prevent a class from being inherited, precede the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final

examples

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

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

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

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