Some code that won't compile

```
public class Point2D {
  private int x, y;
  public Point2D(int x, int y) {
    this.x = x;   this.y = y;
  }
}

public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    this.x = x;  this.y = y;  // can't do this!
    this.z = z;
  }
}
```

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

`super` and constructors

- if the superclass has a constructor that requires any arguments (not `()`), you *must* put a constructor in the subclass and have it call the super-constructor (call to super-constructor must be the first statement)

`super` and constructors

```
public class Point2D {
  private int x, y;

  public Point2D(int x, int y) {
    this.x = x;   this.y = y;
  }
}

public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    super(x, y);   // calls Point2Dconstructor
    this.z = z;
  }
}
```

`super` keyword

- used to refer to superclass (parent) of current class
- can be used to refer to parent class's methods, variables, constructors to call them
  - needed when there is a name conflict with current class
- useful when overriding and you want to keep the old behavior but add new behavior to it

- syntax:

```
super(args);                 // call parent’s constructor
super.fieldName              // access parent’s field
super.methodName(args);      //   or method
```

`super` example

```
public class BankAccount {
  private double myBalance;
  public BankAccount() { myBalance = 0; }
  public double getBalance() { return myBalance; }
  public void withdraw(double amount) {
    myBalance -= amount;
} }

public class FeeAccount
                    extends BankAccount {
  public void withdraw(double amount) {
    super.withdraw(amount);
    if (getBalance() < 100.00)
      withdraw(2.00); // charge $2 fee
} }
```

- didn't need to say `super.getBalance()` because the `FeeAccount` subclass doesn't override that method (it is unambiguous which version of the method we are talking about)
- Can the withdraw(2.00) cause any problems?

Constructors in extended classes

- A constructor of the extended class can invoke one of the superclass's constructors by using the `super` method.
- If no superclass constructor is invoked explicitly, then the superclass's no-arg constructor

  ```
  super( )
  ```

  is invoked automatically as the first statement of the extended class's constructor.
- Constructors are not methods and are NOT inherited.

Three phases of an object's construction

- When an object is created, memory is allocated for all its fields, which are initially set to be their default values. It is then followed by a three-phase construction:
  - invoke a superclass's constructor
  - initialize the fields by using their initializers and initialization blocks
  - execute the body of the constructor

- The invoked superclass's constructor is executed using the same three-phase constructor. This process is executed recursively until the `Object` class is reached

To Illustrate the Construction Order. . .

```
class X {
     protected int xOri = 1;
     protected int whichOri;
     public X() {
          whichOri = xOri;
     }
}
```

```
class Y extends X {
     protected int yOri = 2;
     public Y() {
          whichOri = yOri;
     }
}
```

```
Y objectY = new Y();
```

Step	what happens	xOri	yOri	whichOri
0	fields set to default values	0	0	0
1.	Y constructor invoked	0	0	0
2.	X constructor invoked	0	0	0
3.	Object constructor invoked	0	0	0
4.	X field initialization	1	0	0
5.	X constructor executed	1	0	1
6.	Y field initialization	1	2	1
7.	Y constructor executed	1	2	2

Overloading and Overriding Methods

- *Overloading*: providing more than one method with the same name but different parameter list
  - overloading an inherited method means simply adding new method with the same name and different signature

- *Overriding*: replacing the superclass's implementation of a method with your own design.
  - both the parameter lists and the return types must be exactly the same
  - if an overriding method is invoked on an object of the subclass, then it's the subclass's version of this method that gets implemented
  - an overriding method can have different access specifier from its superclass's version, but only wider accessibility is allowed
  - the overriding method's `throws` clause can have fewer types listed than the method in the superclass, or more specific types

Fields/Methods in Extended Classes

- An object of an extended class contains two sets of variables and methods
  1. fields/methods which are defined locally in the extended class
  2. fields/methods which are inherited from the superclass

?

Accessibility and Overriding

- a method can be overridden only if it’s accessible in the subclass
  - `private` methods in the superclass
    - cannot be overridden
    - if a subclass contains a method which has the same signature as one in its superclass, these methods are totally unrelated
  - `package` methods in the superclass
    - can be overridden if the subclass is in the same package as the superclass
  - `protected`, `public` methods
    - always will be

Not as that simple as it seems!

```
package P1;

public class Base {
   private void pri( ) { System.out.println(“Base.pri()”); }
           void pac( ) { System.out.println(“Base.pac()”); }
   protected void pro( ) { System.out.println(“Base.pro()”); }
   public void pub( ) { System.out.println(“Base.pub()”); }

   public final void show( ) {
       pri();  pac();  pro();  pub(); }
}
```

```
package P2;

import P1.Base;

public class Concrete1 extends Base {
   public void pri( ) { System.out.println(“Concrete1.pri()”); }
   public void pac( ) { System.out.println(“Concrete1.pac()”); }
   public void pro( ) { System.out.println(“Concrete1.pro()”); }
   public void pub( ) { System.out.println(“Concrete1.pub()”); }
}
```

```
Concrete1 c1 = new Concrete1();
c1.show( );
```

Output?

```
Base.pri()
Base.pac()
Concrete1.pro()
Concrete1.pub()
```

Sample classes (cont.)

```
package P1;

import P2.Concrete1;

public class Concrete2 extends Concrete1 {
   public void pri( ) { System.out.println(“Concrete2.pri()”); }
   public void pac( ) { System.out.println(“Concrete2.pac()”); }
   public void pro( ) { System.out.println(“Concrete2.pro()”); }
   public void pub( ) { System.out.println(“Concrete2.pub()”); }
}
```

```
Concrete2 c2 = new Concrete2();
c2.show( );
```

Output?

```
Base.pri()
Concrete2.pac()
Concrete2.pro()
Concrete2.pub()
```

Sample classes (cont.)

```
package P3;
import P1.Concrete2;
public class Concrete3 extends Concrete2 {
   public void pri( ) { System.out.println(“Concrete3.pri()”); }
   public void pac( ) { System.out.println(“Concrete3.pac()”); }
   public void pro( ) { System.out.println(“Concrete3.pro()”); }
   public void pub( ) { System.out.println(“Concrete3.pub()”); }
}
```

```
Concrete3 c3 = new Concrete3();
c3.show( );
```

Output?

```
Base.pri()
Concrete3.pac()
Concrete3.pro()
Concrete3.pub()
```

Hiding fields

- Fields cannot be overridden, they can only be hidden
- If a field is declared in the subclass and it has the same name as one in the superclass, then the field belongs to the superclass cannot be accessed directly by its name any more

The `instanceof` keyword

- Performs run-time type check on the object referred to by a reference variable
- Usage: *object-reference* `instanceof` *type*
  (result is a boolean expression)
  - if *type* is a class, evaluates to true if the variable refers to an object of *type* or any subclass of it.
  - if *type* is an interface, evaluates to true if the variable refers to an object that implements that interface.
  - if *object-reference* is null, the result is false.
- Example:

```
Object o = myList.get(2);
if (o instanceof BankAccount)
  ((BankAccount)o).deposit(10.0);
```

Polymorphism

- An object of a given class can have multiple forms: either as its declared class type, or as any subclass of it
- an object of an extended class can be used wherever the original class is used
- **Question**: given the fact that an object’s actual class type may be different from its declared type, then when a method accesses an object’s member which gets redefined in a subclass, then which member the method refers to (subclass’s or superclass’s)?
  - when you invoke a method through an object reference, the *actual class of the object* decides which implementation is used
  - when you access a field, the declared type of the reference decides which implementation is used

Output?

```
Extend.show: ExtendStr
Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

```
class SuperShow {
   public String str = "SuperStr";

   public void show( ) {
        System.out.println("Super.show:" + str);
   }
}

class ExtendShow extends SuperShow {
   public String str = "ExtendedStr";

   public void show( ) {
        System.out.println("Extend.show:" + str);
   }

   public static void main (String[] args) {
      ExtendShow ext = new ExtendShow( );
      SuperShow sup = ext;
      sup.show( );   //1
      ext.show( );   //2      methods invoked through object reference
      System.out.println("sup.str = " + sup.str);      //3
      System.out.println("ext.str = " + ext.str);//4    field access
   }
}
```