Inheritance & Polymorphism

University of Wisconsin
LA CROSSE™

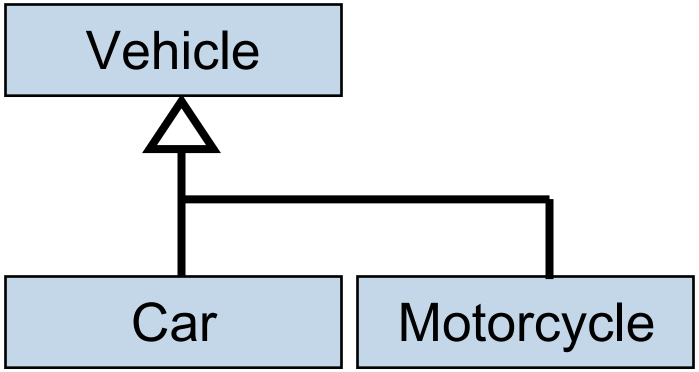
Common Features in Classes

- **Certain types of objects have things in common**
  - Cars, trucks, motorcycles
  - Savings, checking, investment accounts

- **In Java, such similarities are exploited by inheritance**
  - **Inheritance** is a way of writing common code once, and using it in many classes
    - Code can be made **simpler** and **more useful**
  - Similarities are written into the **super-class** (parent)
  - Differences are written into the **sub-classes** (children)

Inheritance

- **Software reuse is at the heart of inheritance**
- **The sub-class inherits all properties of the parent**
  - All methods
  - All class variables
- **Inheritance relationships can be represented as a diagram, with arrows from children to parents**



Syntax

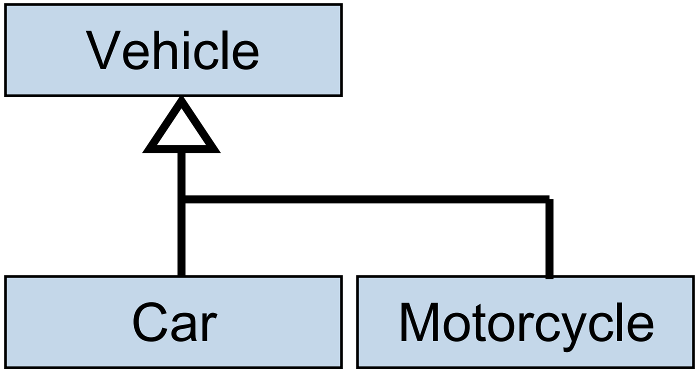
```
public class SuperClass {
   // data and methods
}
```

```
public class SubClass extends SuperClass {
   // more data and methods
}
```

Inheritance

UML: Unified Modeling Language

- Software reuse is at the heart of inheritance
- Inheritance relationships can be represented as a diagram, with arrows from children to the parent



UML Diagram indicating inheritance

```
public class Vehicle {
   // data and methods
}
```

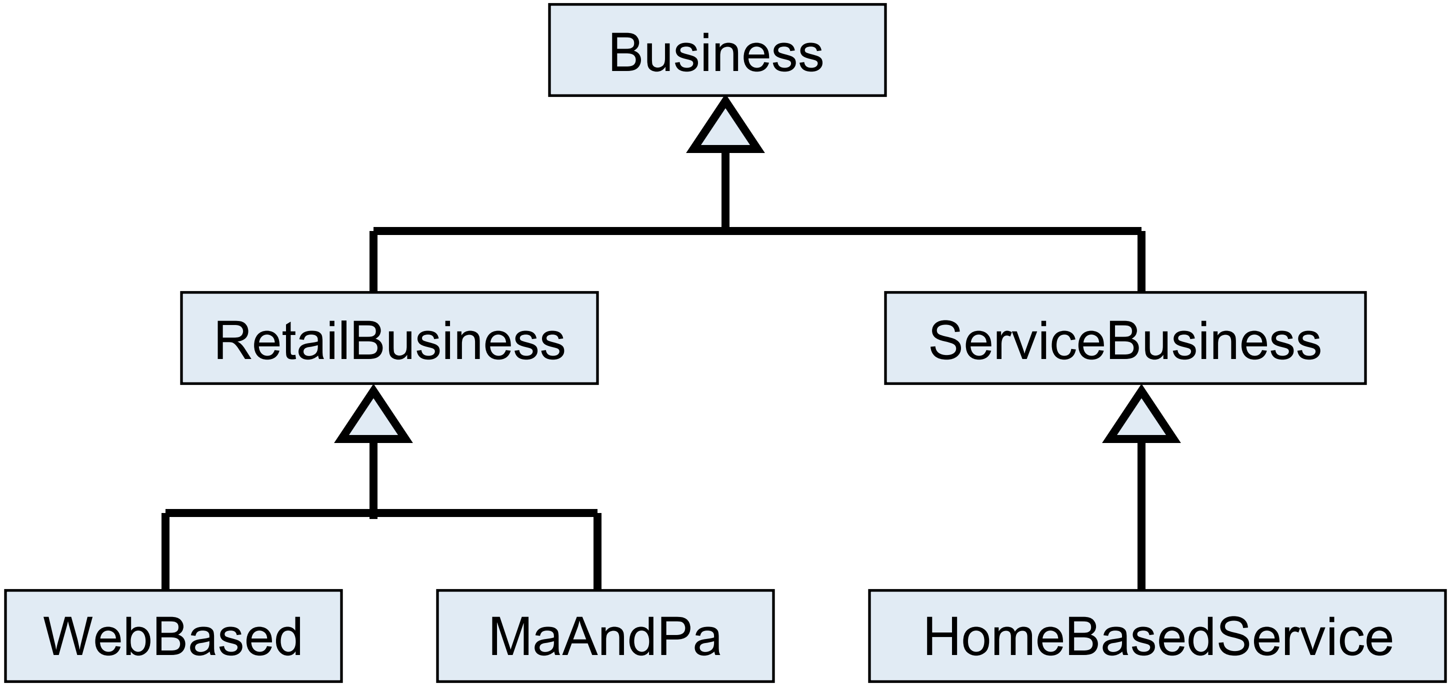
```
public class Car extends Vehicle {
   // more data and methods
}
```

```
public class Motorcycle extends Vehicle {
   // more data and methods
}
```

Java code indicating inheritance

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a class hierarchy



Constructors and the super Reference

- All methods and variables of a parent are inherited, except for the constructor method
- A child's constructor is responsible for calling the parent constructor

- The reserved keyword: super
  - can be used as a method to call the parent's constructor

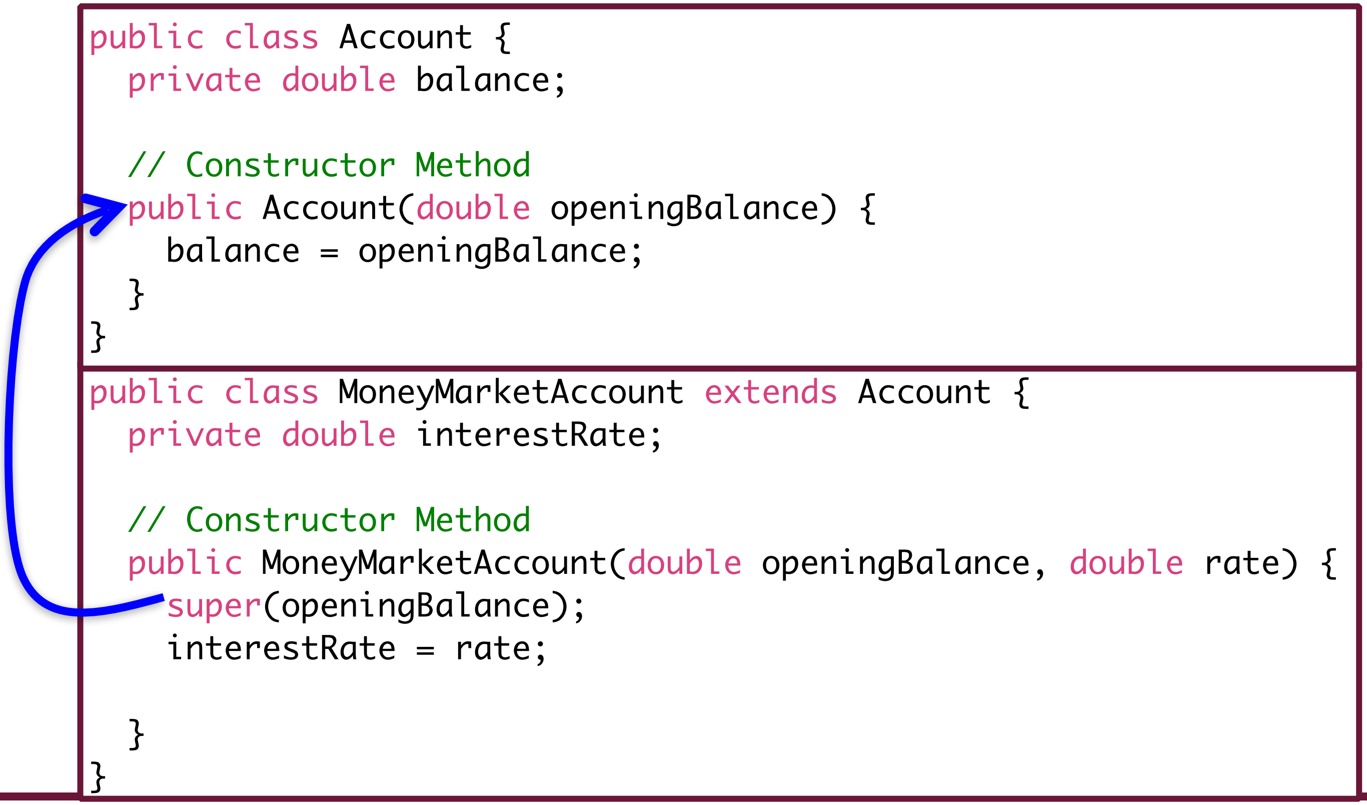
```
super();
super( arg1, arg2, ... );
```

  - can be used as a direct reference to the parent class
    - Often optional since the child can call the class members directly

```
super.methodName();
super.variableIdentifer;
```

Running the super Constructor

- **General Rule:**
  - The child **must** call the parent's constructor **before** initializing itself.



```
public class Account {
  private double balance;

  // Constructor Method
  public Account(double openingBalance) {
    balance = openingBalance;
  }
}
```

```
public class MoneyMarketAccount extends Account {
  private double interestRate;

  // Constructor Method
  public MoneyMarketAccount(double openingBalance, double rate) {
    super(openingBalance);
    interestRate = rate;

  }
}
```

Visibility and Inheritance

So what if we only want the children to access a class member of the parent?

Sub-classes cannot access class members marked as private

```
public class Account {
  private double balance;

  // Constructor Method
  public Account(double openingBalance) {
    balance = openingBalance;
  }
}
```

```
public class MoneyMarketAccount extends Account {
  private double interestRate;

  // Constructor Method
  public MoneyMarketAccount(double openingBalance, double rate) {
    super(openingBalance);
    interestRate = rate;
    System.out.println("Balance: " + balance);
  }
}
```

This will not work!

Visibility and Inheritance

- **Fix visibility problem by either:**
  - Creating **public** methods for access, or
  - Using the **protected** scope
    - All sub-classes can directly access (**private** in all other classes)

```
public class Account {
  protected double balance;

  // Constructor Method
  public Account(double openingBalance) {
    balance = openingBalance;
  }
}
```

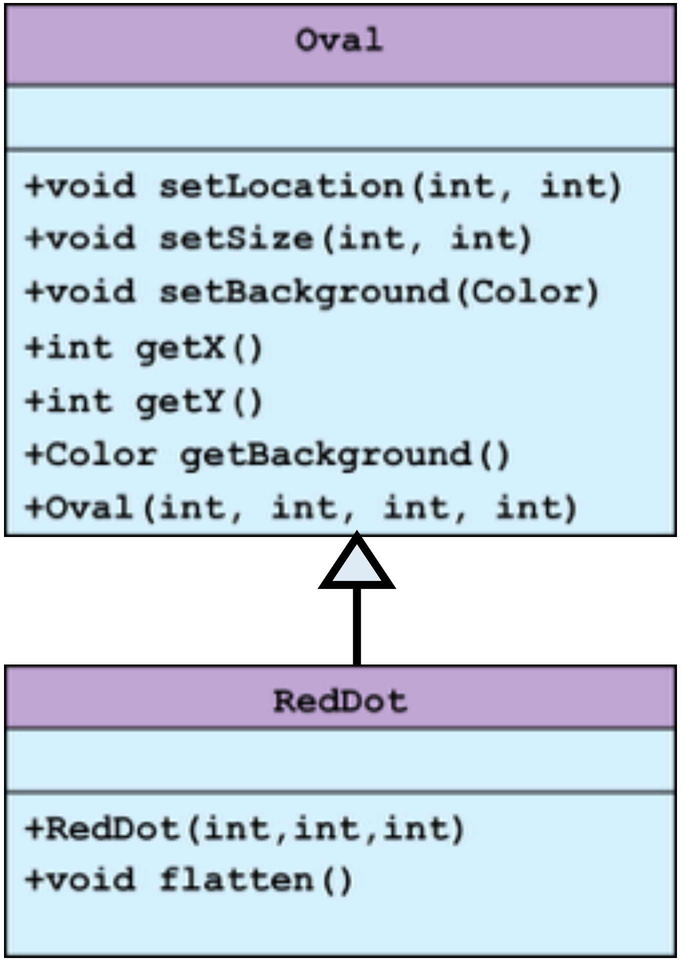
```
public class MoneyMarketAccount extends Account {
  private double interestRate;

  // Constructor Method
  public MoneyMarketAccount(double openingBalance, double rate) {
    super(openingBalance);
    interestRate = rate;
    System.out.println("Balance: " + balance);
  }
}
```

This will work!

Inheritance Example

```java
import java.awt.Color;

public class RedDot extends Oval {
   // Constructor Method
   public RedDot(int x, int y, int d) {
      super(x, y, d, d);
      setBackground( Color.RED );
   }

   public void flatten() {
      setSize( getWidth()+10, getHeight()-10);
      repaint();
   }
}
```

```java
public class Driver {
   // ...
   public Driver() {
      Window window  = new Window();
      // Setup the window...

      RedDot dotty = new RedDot(10, 20, 50);
      dotty.setLocation(20, 30);
      dotty.flatten();
      window.add(dotty);
   }
}
```



Oval

+void setLocation(int, int)
+void setSize(int, int)
+void setBackground(Color)
+int getX()
+int getY()
+Color getBackground()
+Oval(int, int, int, int)

RedDot

+RedDot(int,int,int)
+void flatten()

Polymorphism: Overriding Methods

- **If we want to replace a method defined by the parent class we can create a new one in the child class to override it.**
  - Must have the exact same method signature:
    - Exact same access, name, list of parameters, and return type

- **The sub-class method is able to re-define the behavior of the super-class method**

Polymorphism

The occurrence of something (method) in several different forms.

Allows us to easily modify and/or extend existing functionality.

- Overriding

  We are focusing here at the moment.

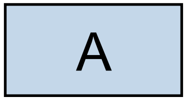
  - The sub-class can **replace a method** inherited from the super-class.
  - **Must have the exact same method signature**:
    - Exact same access, name, list of parameters, and return type
- Overloading

  We will return to this later.

  - Different parameters determine which implementation of the **same method name** is used.
  - Must have the exact same method name, BUT can have different types for the parameters, and different numbers of parameters.

Polymorphism:
Overriding Methods Example

```
public class A {
  protected int i, j;

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

  public void show() {
    System.out.println("("+i+", "+j+")");
  }
}
```



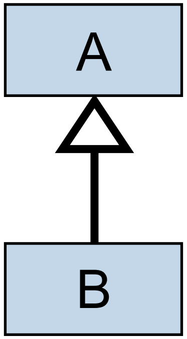
```
public class Example {
  public Example() {




    A anAThing = new A(4, 5);

    anAThing.show();
  }
}
```

```
(4, 5)
```

Polymorphism: Overriding Methods Example

```
public class A {
  protected int i, j;

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

  public void show() {
    System.out.println("("+i+", "+j+")");
  }
}
```

```
public class B extends A {
  protected int k;

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

  public void show() {
    System.out.println("("+i+", "+j+", "+k+")");
  }
}
```



```
public class Example {
  public Example() {
    B aBThing = new B(1, 2, 3);

    aBThing.show();

    A anAThing = new A(4, 5);

    anAThing.show();
  }
}
```

```
(1, 2, 3)
(4, 5)
```

A few (more) things to know about Objects in Java

Object

- **All classes descend from the java.lang.Object class**

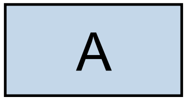
- **Two methods that should be overridden**
  - `.equals(other)` returns true if the calling object is equal to the other object, and false otherwise.

    ```
    public boolean equals(Object other)
    ```

  - `.toString()` returns a textual representation of the calling object

    ```
    public String toString()
    ```

String toString()



A

```
public class A {
  protected int i, j;

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

  public void show() {
    System.out.println("("+i+", "+j+")");
  }





}
```

```
public class Example {
  public Example() {
    A anAThing = new A(4, 5);

    anAThing.show();

    // The two statements below both call
    // the toString method of the object
    System.out.println( anAThing );
    System.out.println( anAThing.toString() );
  }
}
```

```
(4, 5)
A@27ecfcd9
A@27ecfcd9
```

String toString()

A

```
public class A {
  protected int i, j;

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

  public void show() {
    System.out.println("("+i+", "+j+")");
  }

  public String toString() {
    String output = "In toString ";
    output += "["+i+", "+j+"]";
    return output;
  }
}
```

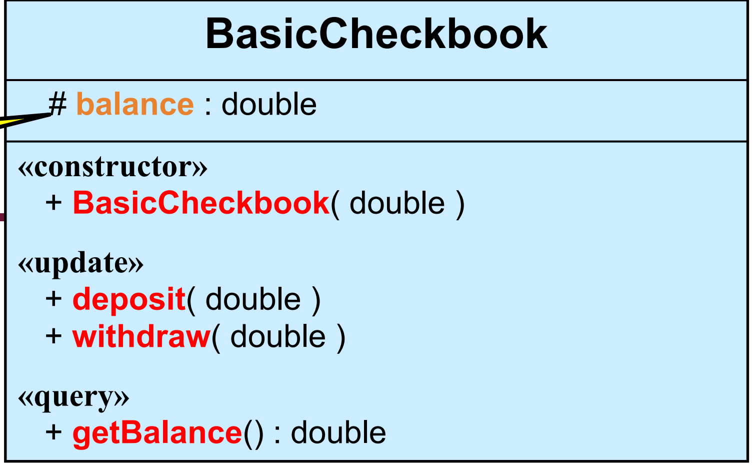
```
public class Example {
  public Example() {
    A anAThing = new A(4, 5);

    anAThing.show();

    // The two statements below both call
    // the toString method of the object
    System.out.println( anAThing );
    System.out.println( anAThing.toString() );
  }
}
```

```
(4, 5)
In toString [4, 5]
In toString [4, 5]
```

Now back to Inheritance

Inheritance Example

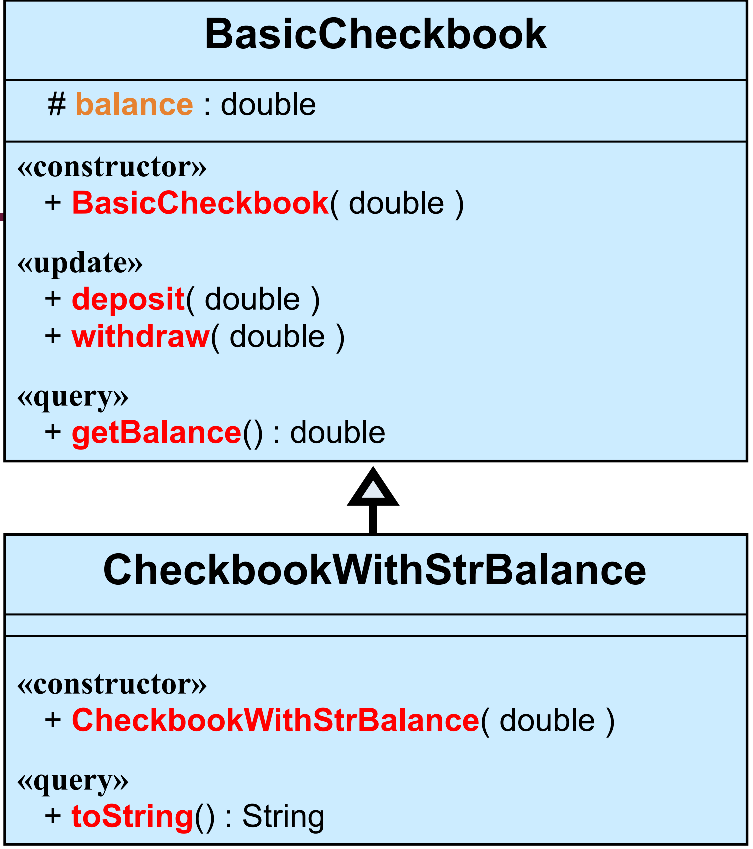
```
public class BasicCheckbook {
  protected double balance;

  public BasicCheckbook(double cash) {
    balance = cash;
  }

  public void deposit(double cash) {
    balance = balance + cash;
  }

  public void withdraw(double cash) {
    balance = balance - cash;
  }
  public double getBalance() {
    return balance;
  }
}
```



# means protected

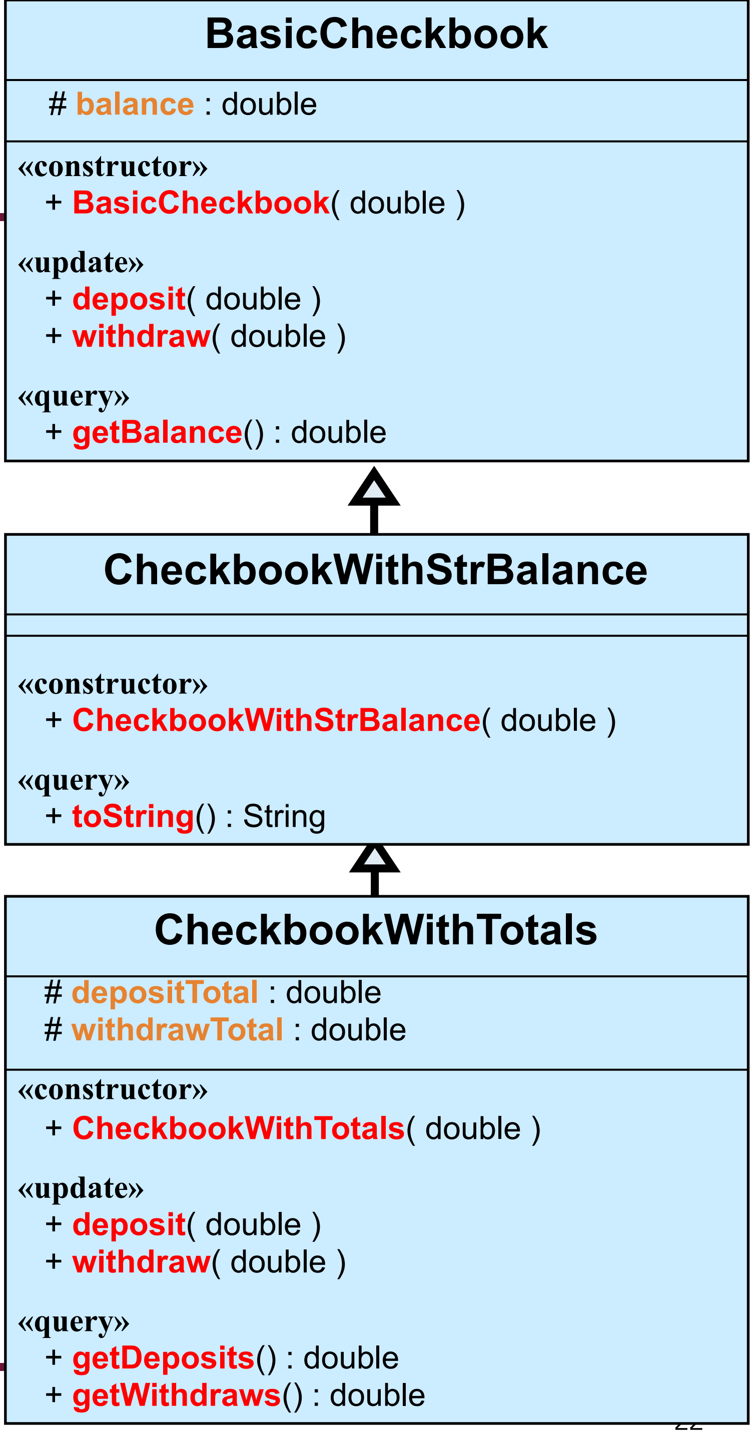
Inheritance Example

```java
public class CheckbookWithStrBalance
             extends BasicCheckbook {

  public CheckbookWithStrBalance(double cash) {
    super(cash);
  }

  public String toString() {
    DecimalFormat df = new DecimalFormat("0.00");
    return "$"+ df.format(balance);
  }
}
```



Inheritance Example

```java
public class CheckbookWithTotals
             extends CheckbookWithStrBalance {
  protected double depositTotal, withdrawTotal;

  public CheckbookWithTotals(double cash) {
    super(cash);
    depositTotal = 0.0;
    withdrawTotal = 0.0;
  }

  public void deposit(double cash) {
    super.deposit(cash);
    depositTotal = depositTotal + cash;
  }

  public void withdraw(double cash) {
    super.withdraw(cash);
    withdrawTotal = withdrawTotal + cash;
  }

  public double getDeposits() {
    return depositTotal;
  }

  public double getWithdraws() {
    return withdrawTotal;
  }
}
```



BasicCheckbook

# balance : double

«constructor»
+ BasicCheckbook(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double

CheckbookWithStrBalance

«constructor»
+ CheckbookWithStrBalance(double)

«query»
+ toString() : String

CheckbookWithTotals

# depositTotal : double
# withdrawTotal : double

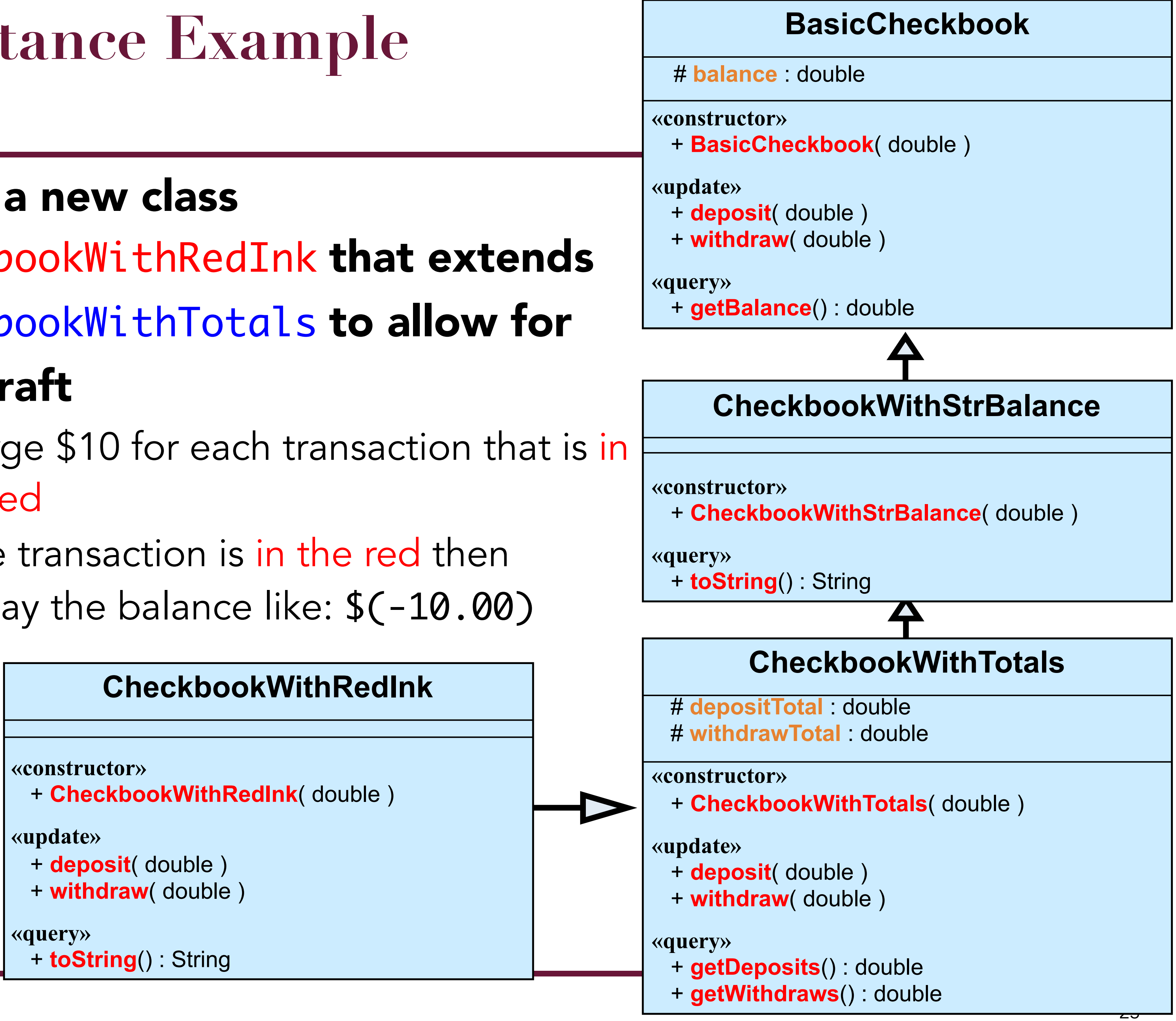
«constructor»
+ CheckbookWithTotals(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getDeposits() : double
+ getWithdraws() : double

Inheritance Example

- **Write a new class** `CheckbookWithRedInk` **that extends** `CheckbookWithTotals` **to allow for overdraft**
  - Charge $10 for each transaction that is in the red
  - If the transaction is in the red then display the balance like: `$(-10.00)`



BasicCheckbook

# balance : double

«constructor»
+ BasicCheckbook(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double

CheckbookWithStrBalance

«constructor»
+ CheckbookWithStrBalance(double)

«query»
+ toString() : String

CheckbookWithTotals

# depositTotal : double
# withdrawTotal : double

«constructor»
+ CheckbookWithTotals(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getDeposits() : double
+ getWithdraws() : double

CheckbookWithRedInk

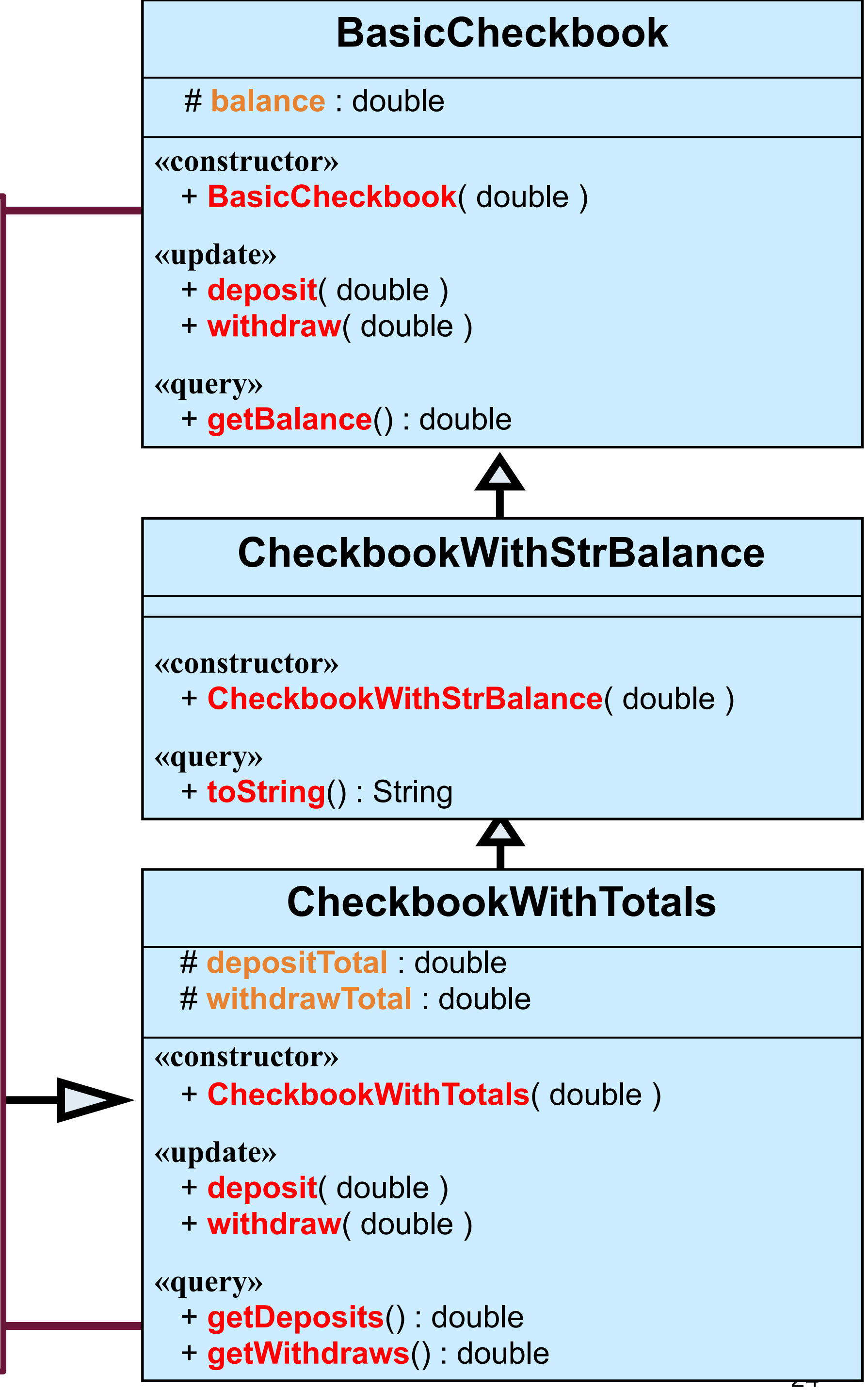
«constructor»
+ CheckbookWithRedInk(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ toString() : String

Inheritance Example

```java
public class CheckbookWithRedInk
             extends CheckbookWithTotals {
  public CheckbookWithRedInk(double cash) {
    super(cash);
  }
  public void deposit(double cash) {
    super.deposit(cash);
    if( balance < 0 ) {
      System.out.println("$10 surcharge");
      balance = balance - 10;
    }
  }
  public void withdraw(double cash) {
    super.withdraw(cash);
    if( balance < 0 ) {
      System.out.println("$10 surcharge");
      balance = balance - 10;
    }
  }
  public String toString () {
    DecimalFormat df = new DecimalFormat("0.00");
    if( balance >= 0 ) {
      return super.toString();
    } else {
      return "$("+ df.format(balance) + ")";
    }
  }
}
```



BasicCheckbook

# balance : double

«constructor»
+ BasicCheckbook(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double

CheckbookWithStrBalance

«constructor»
+ CheckbookWithStrBalance(double)

«query»
+ toString() : String

CheckbookWithTotals

# depositTotal : double
# withdrawTotal : double

«constructor»
+ CheckbookWithTotals(double)

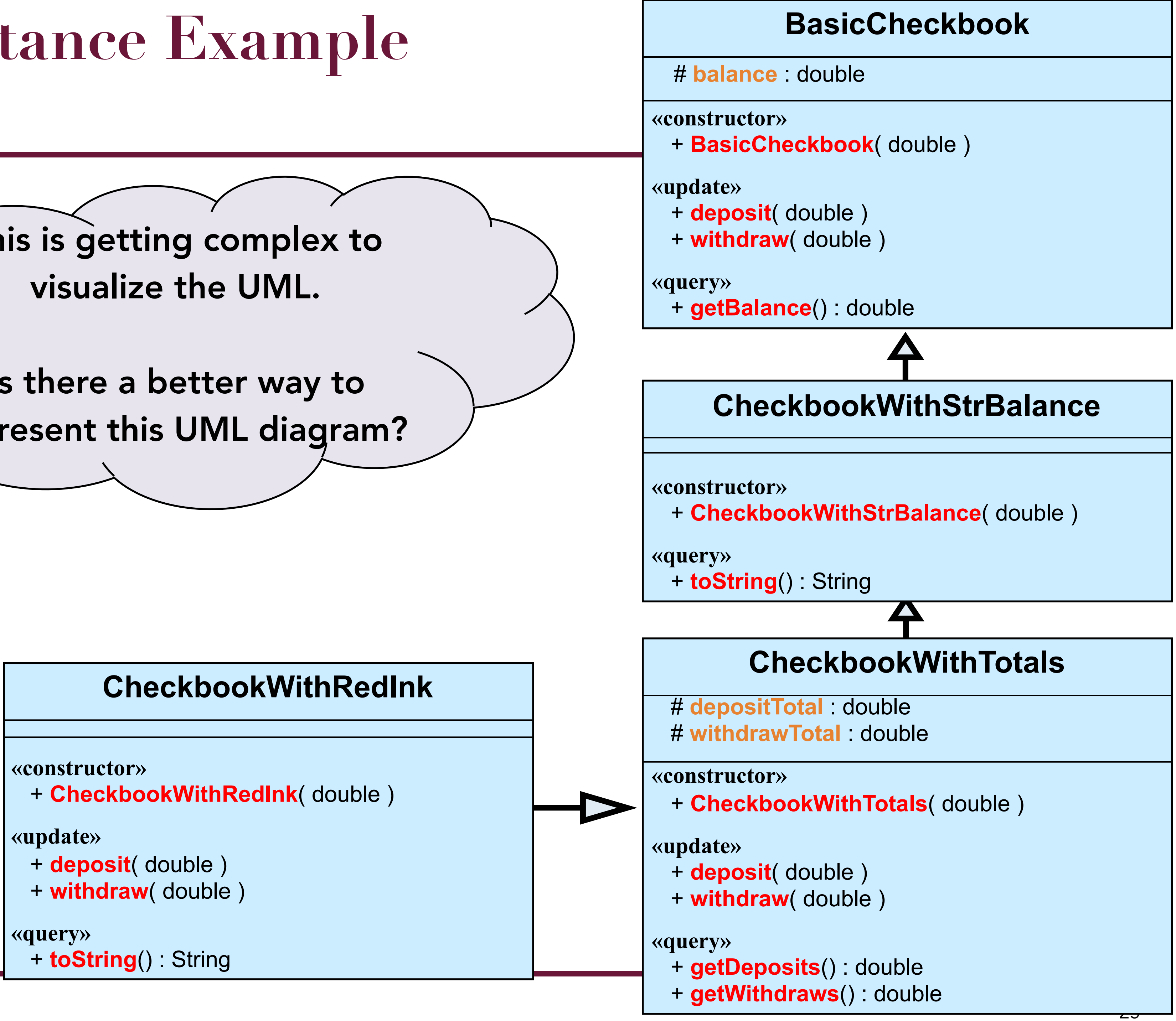
«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getDeposits() : double
+ getWithdraws() : double

Inheritance Example

This is getting complex to visualize the UML.

Is there a better way to represent this UML diagram?



BasicCheckbook

# balance : double

«constructor»
+ BasicCheckbook(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double

CheckbookWithStrBalance

«constructor»
+ CheckbookWithStrBalance(double)

«query»
+ toString() : String

CheckbookWithTotals

# depositTotal : double
# withdrawTotal : double

«constructor»
+ CheckbookWithTotals(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getDeposits() : double
+ getWithdraws() : double

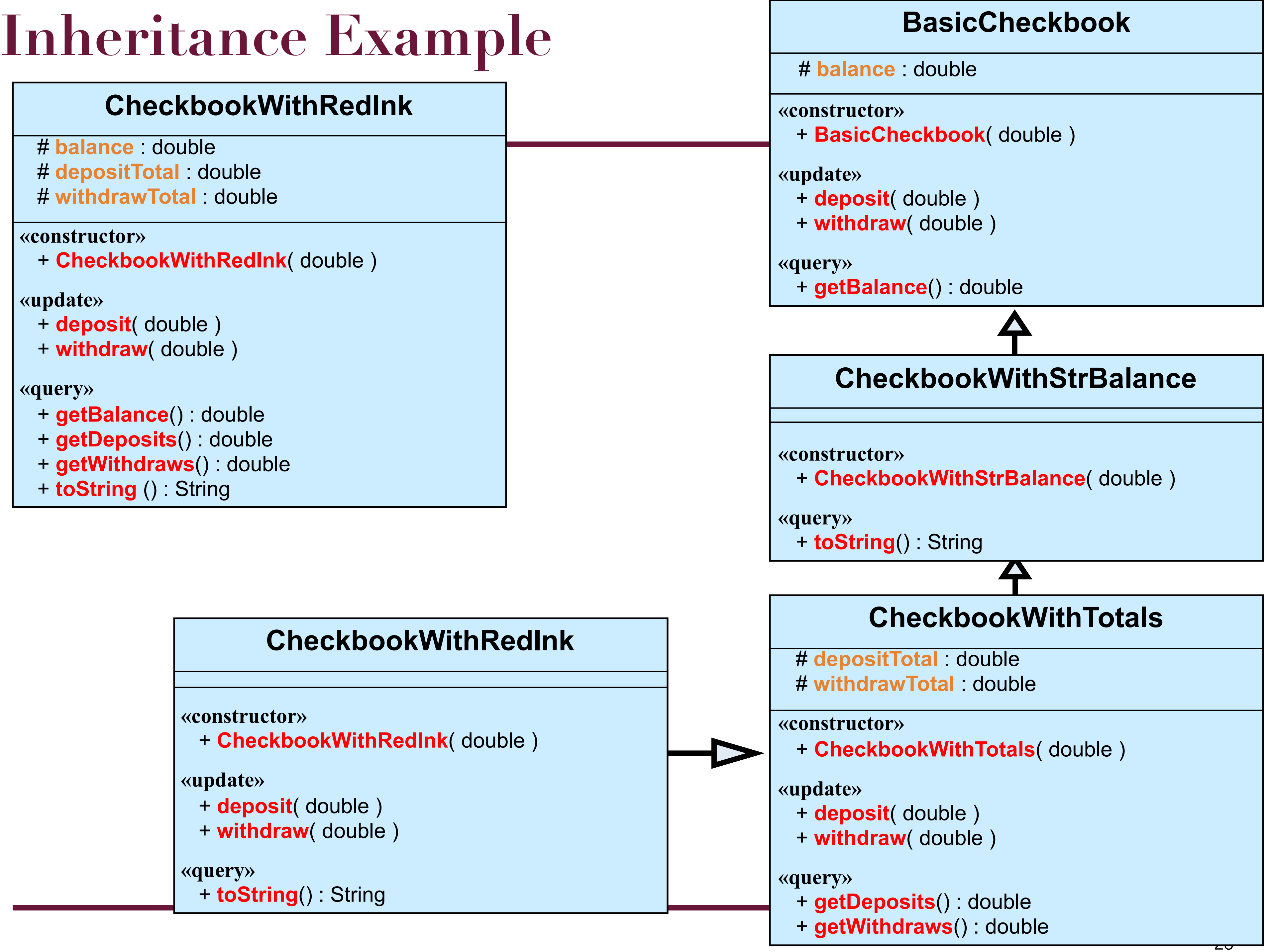
CheckbookWithRedInk

«constructor»
+ CheckbookWithRedInk(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ toString() : String

Inheritance Example



CheckbookWithRedInk

# balance : double
# depositTotal : double
# withdrawTotal : double

«constructor»
+ CheckbookWithRedInk(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double
+ getDeposits() : double
+ getWithdraws() : double
+ toString () : String

BasicCheckbook

# balance : double

«constructor»
+ BasicCheckbook(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double

CheckbookWithStrBalance

«constructor»
+ CheckbookWithStrBalance(double)

«query»
+ toString() : String

CheckbookWithTotals

# depositTotal : double
# withdrawTotal : double

«constructor»
+ CheckbookWithTotals(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getDeposits() : double
+ getWithdraws() : double

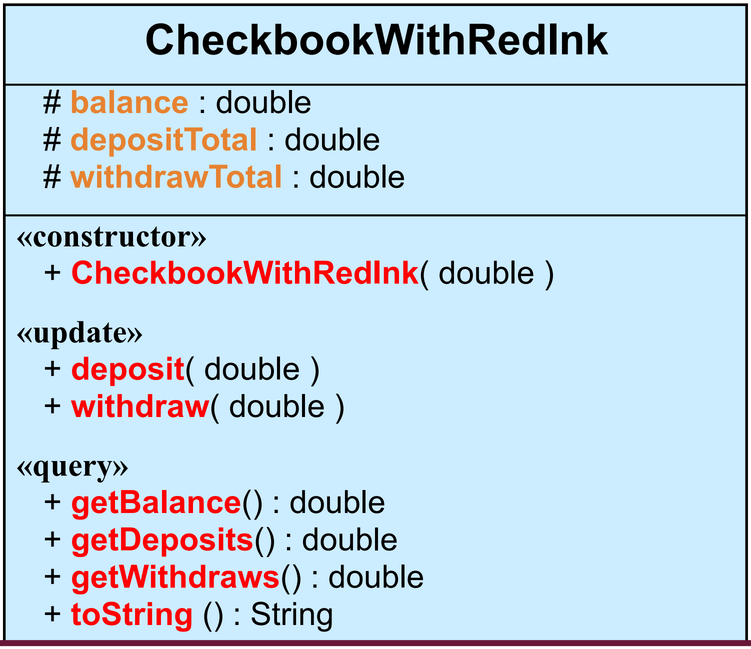
CheckbookWithRedInk

«constructor»
+ CheckbookWithRedInk(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ toString() : String

Inheritance Example
Flattened Class Diagram



CheckbookWithRedInk

# balance : double
# depositTotal : double
# withdrawTotal : double

«constructor»
+ CheckbookWithRedInk(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double
+ getDeposits() : double
+ getWithdraws() : double
+ toString () : String

```
public class Driver {
  private CheckbookWithRedInk checkbook;

  public Driver( ) {
    checkbook = new CheckbookWithRedInk( 100.00 );
    checkbook.deposit( 20.00 );
    checkbook.withdraw( 125.99 );
    System.out.println("Final Balance: "+ checkbook.toString());
  }
}
```

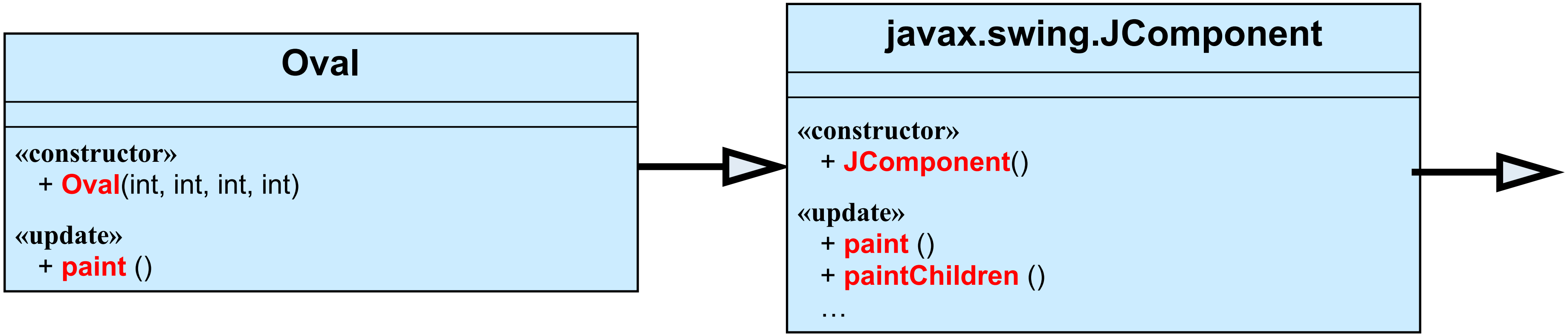
```
Final Balance: $(-5.99)
```

Inheritance Example: Oval and JComponent

```
public class Oval extends JComponent {

  public Oval( int x, int y, int w, int h ) {
    super();
    setBounds( x, y, w, h );
    setBackground( Color.black );
  }

  public void paint( Graphics g ) {
    g.setColor( getBackground() );
    g.fillOval( 0, 0, getWidth(), getHeight() );
    paintChildren( g );
  }
}
```

`paint()` **method creates graphics on-screen.**

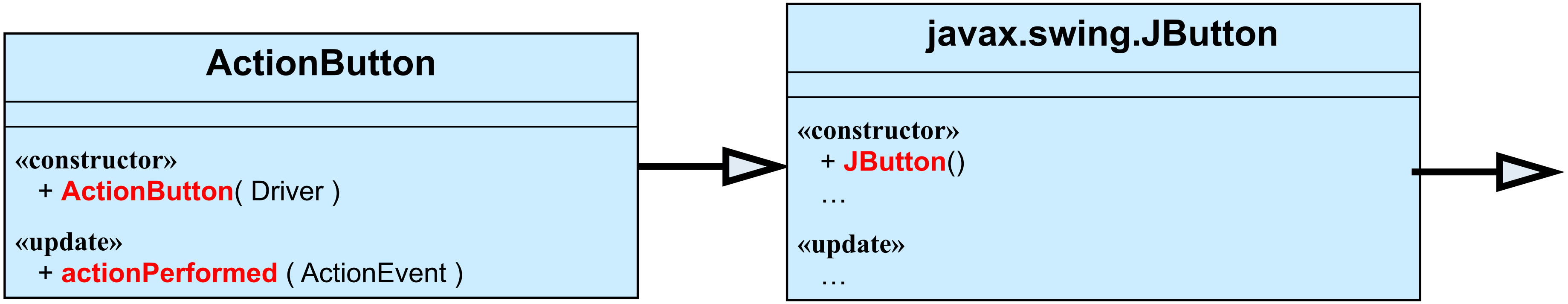
We override `paint()` **to produce our own effect**



Abstract Example: ActionButton and JButton

```
public class ActionButton extends JButton implements ActionListener {
  /** Driver to tell about any action events. */
  private Driver driver;

  public ActionButton( Driver d ) {
    super();
    driver = d;
    addActionListener( this );
  }

  public void actionPerformed( ActionEvent e ) {
    driver.handleButtonAction( this );
  }
```

`actionPerformed()` is an implementation of an **abstract** method



ActionButton

«constructor»
+ **ActionButton**(Driver)

«update»
+ **actionPerformed** (ActionEvent)

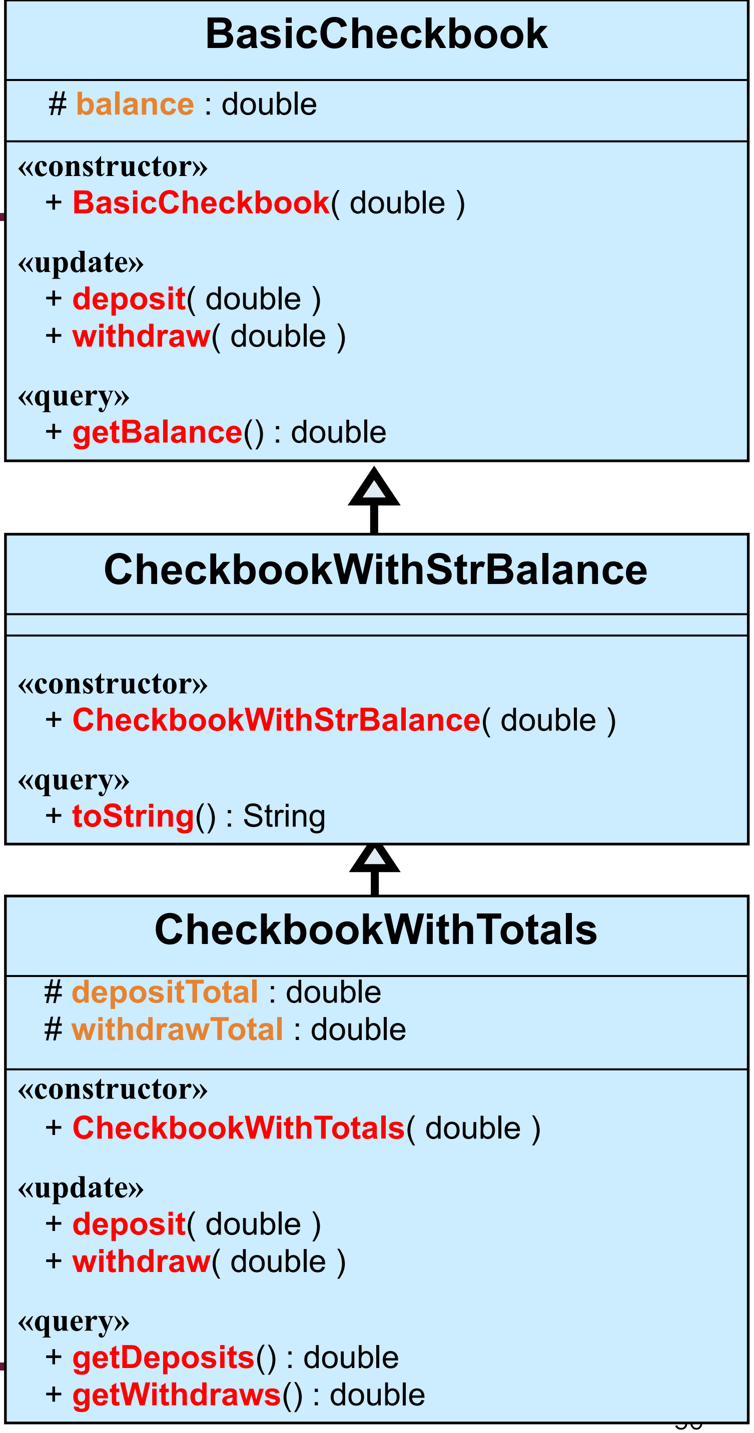
javax.swing.JButton

«constructor»
+ **JButton**()
…

«update»
…

Inheritance Example

```
public class CheckbookWithTotals
             extends CheckbookWithStrBalance {
  protected double depositTotal, withdrawTotal;

  public CheckbookWithTotals(double cash) {
    super(cash);
    depositTotal = 0.0;
    withdrawTotal = 0.0;
  }

  public void deposit(double cash) {
    super.deposit(cash);
    depositTotal = depositTotal + cash;
  }

  public void withdraw(double cash) {
    super.withdraw(cash);
    withdrawTotal = withdrawTotal + cash;
  }

  public double getDeposits() {
    return depositTotal;
  }

  public double getWithdraws() {
    return withdrawTotal;
  }
}
```



BasicCheckbook

# balance : double

«constructor»
+ BasicCheckbook(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double

CheckbookWithStrBalance

«constructor»
+ CheckbookWithStrBalance(double)

«query»
+ toString() : String

CheckbookWithTotals

# depositTotal : double
# withdrawTotal : double

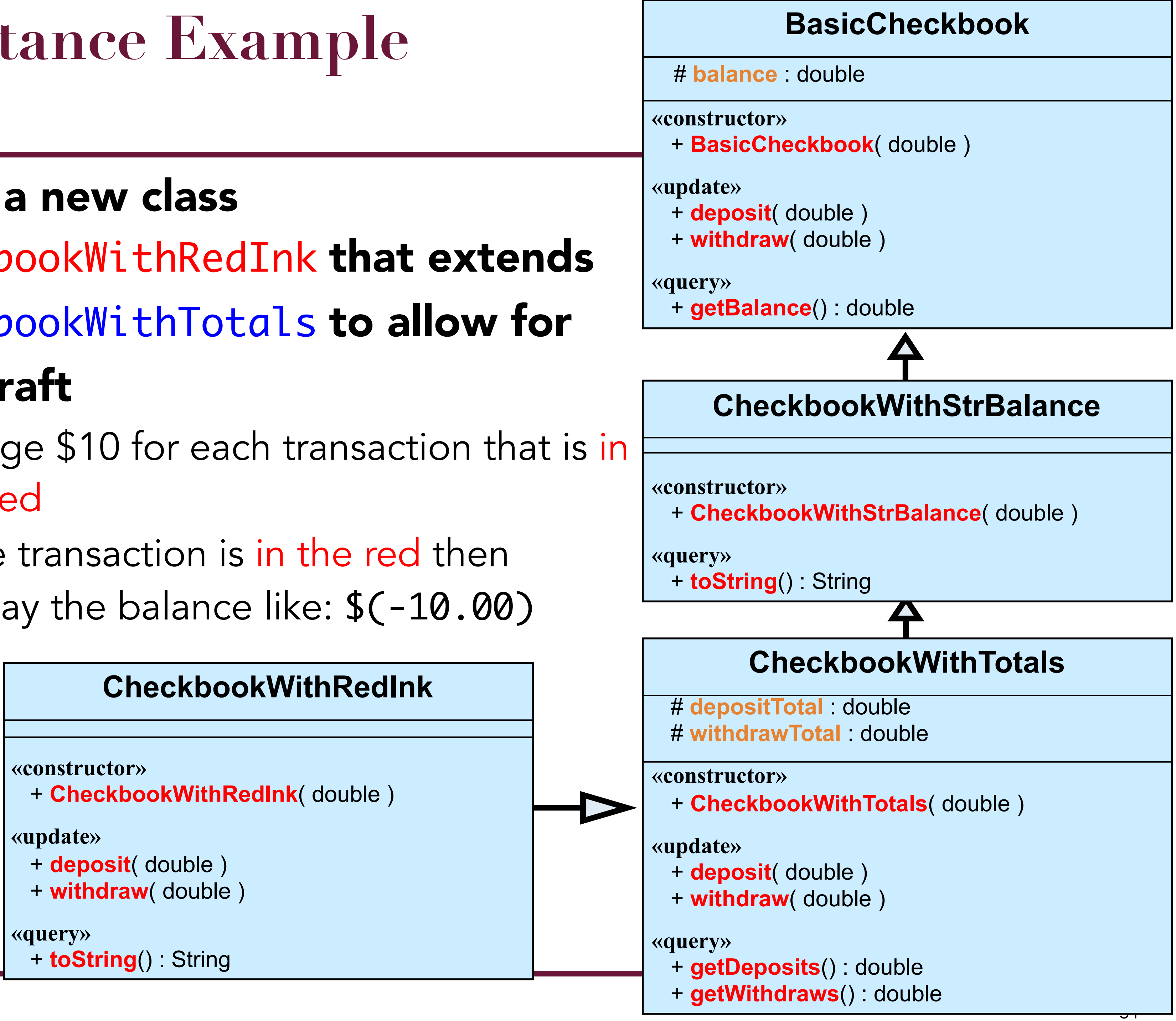
«constructor»
+ CheckbookWithTotals(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getDeposits() : double
+ getWithdraws() : double

Inheritance Example

- **Write a new class** CheckbookWithRedInk **that extends** CheckbookWithTotals **to allow for overdraft**
  - Charge $10 for each transaction that is in the red
  - If the transaction is in the red then display the balance like: `$(-10.00)`



BasicCheckbook

# balance : double

«constructor»
+ BasicCheckbook(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double

CheckbookWithStrBalance

«constructor»
+ CheckbookWithStrBalance(double)

«query»
+ toString() : String

CheckbookWithTotals

# depositTotal : double
# withdrawTotal : double

«constructor»
+ CheckbookWithTotals(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getDeposits() : double
+ getWithdraws() : double

CheckbookWithRedInk

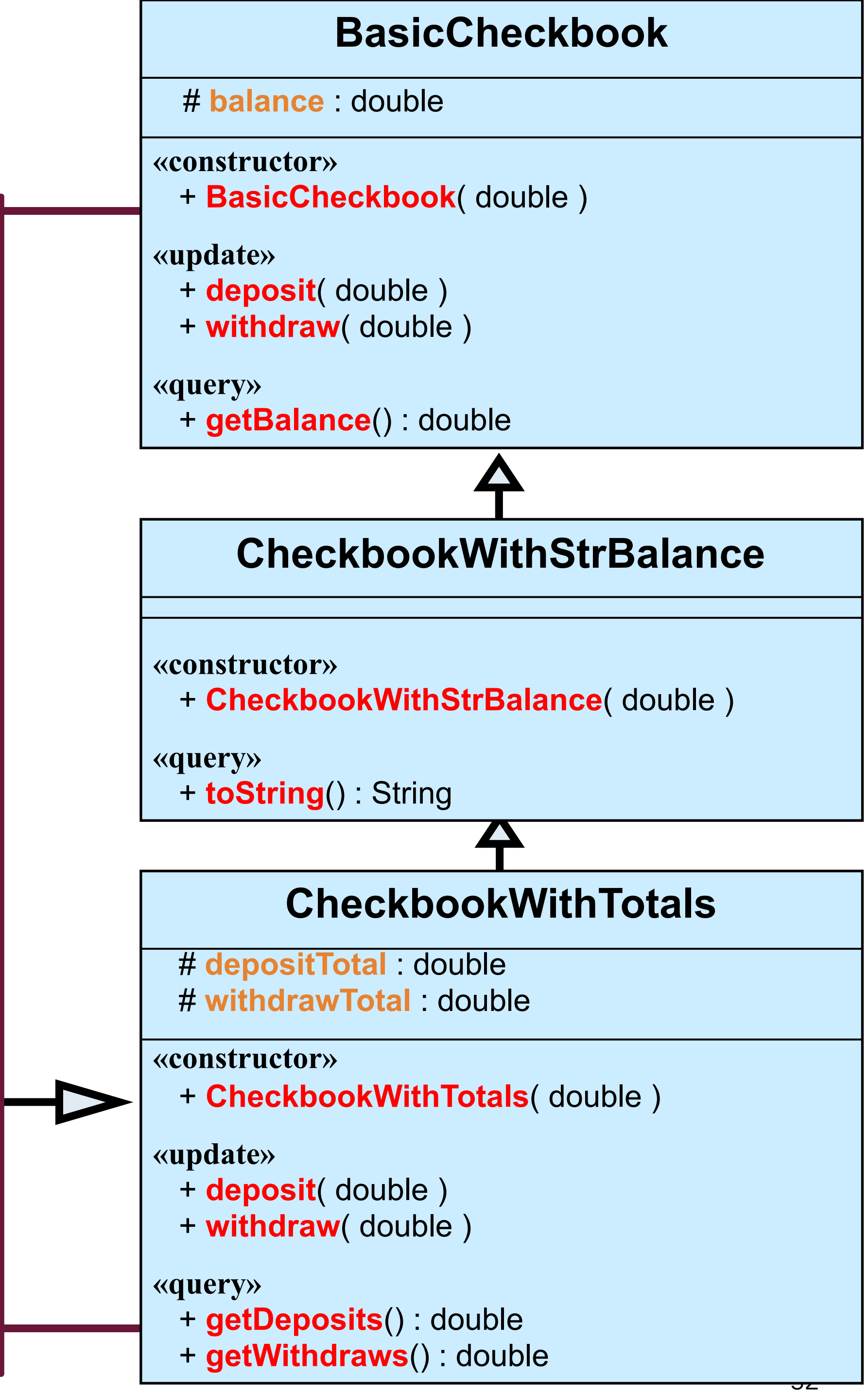
«constructor»
+ CheckbookWithRedInk(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ toString() : String

Inheritance Example

```
public class CheckbookWithRedInk
             extends CheckbookWithTotals {
  public CheckbookWithRedInk(double cash) {
    super(cash);
  }
  public void deposit(double cash) {
    super.deposit(cash);
    if( balance < 0 ) {
      System.out.println("$10 surcharge");
      balance = balance - 10;
    }
  }
  public void withdraw(double cash) {
    super.withdraw(cash);
    if( balance < 0 ) {
      System.out.println("$10 surcharge");
      balance = balance - 10;
    }
  }
  public String toString () {
    DecimalFormat df = new DecimalFormat("0.00");
    if( balance >= 0 ) {
      return super.toString();
    } else {
      return "$("+ df.format(balance) + ")";
    }
  }
}
```



BasicCheckbook

# balance : double

«constructor»
+ BasicCheckbook(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getBalance() : double

CheckbookWithStrBalance

«constructor»
+ CheckbookWithStrBalance(double)

«query»
+ toString() : String

CheckbookWithTotals

# depositTotal : double
# withdrawTotal : double

«constructor»
+ CheckbookWithTotals(double)

«update»
+ deposit(double)
+ withdraw(double)

«query»
+ getDeposits() : double
+ getWithdraws() : double

Type Conformance

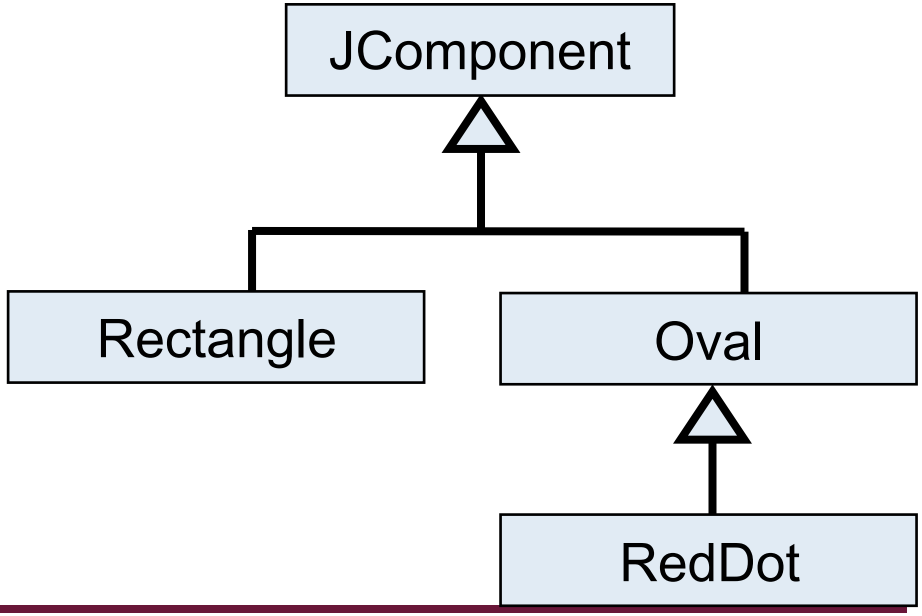
Answering the question:
Is the left-hand side an ancestor of the right-hand side?

- **When performing assignment:**

  x = y;

  - y must conform to x
    - Objects **conform** to the types of their **ancestors**
  - If x and y are **primitives** then the type of y must
    - **be identical** to the type of x, or
    - **widen** to the type of x
  - Otherwise the **class** of y must
    - **be identical** to the class of x, or
    - **be a subclass** of x

```
Oval thing1 = new Oval(10, 10, 40, 50);
RedDot thing2 = new RedDot(10, 10, 100);
JComponent anyThing;

anyThing = thing1; // Correct
thing2 = thing1;   // Incorrect
thing1 = thing2;   // Correct
```



Type Conformance & Overriding Example

```
public class A {
  public A() {
    ;
  }
  public void show() {
    System.out.println("Inside A");
  }
}
```

```
public class B extends A {
  public B() {
    super();
  }
  public void show() {
    System.out.println("Inside B");
  }
}
```

```
public class C extends B {
  public C() {
    super();
  }
  public void show() {
    System.out.println("Inside C");
  }
}
```

```
public class Example {
  public Example() {
    A aThing = new A();
    B bThing = new B();
    C cThing = new C();
    A reference;

    reference = aThing;
    reference.show();

    reference = bThing;
    reference.show();

    reference = cThing;
    reference.show();
  }
}
```

This is ok since B is a subclass of A

```
Inside A
Inside B
Inside C
```

Type Conformance & Overriding Example

```
public class A {
  public A() {
    ;
  }
  public void show() {
    System.out.println("Inside A");
  }
}
```

```
public class B extends A {
  public B() {
    super();
  }
  public void show() {
    System.out.println("Inside B");
  }
}
```

```
public class C extends B {
  public C() {
    super();
  }
  public void show() {
    System.out.println("Inside C");
  }
}
```

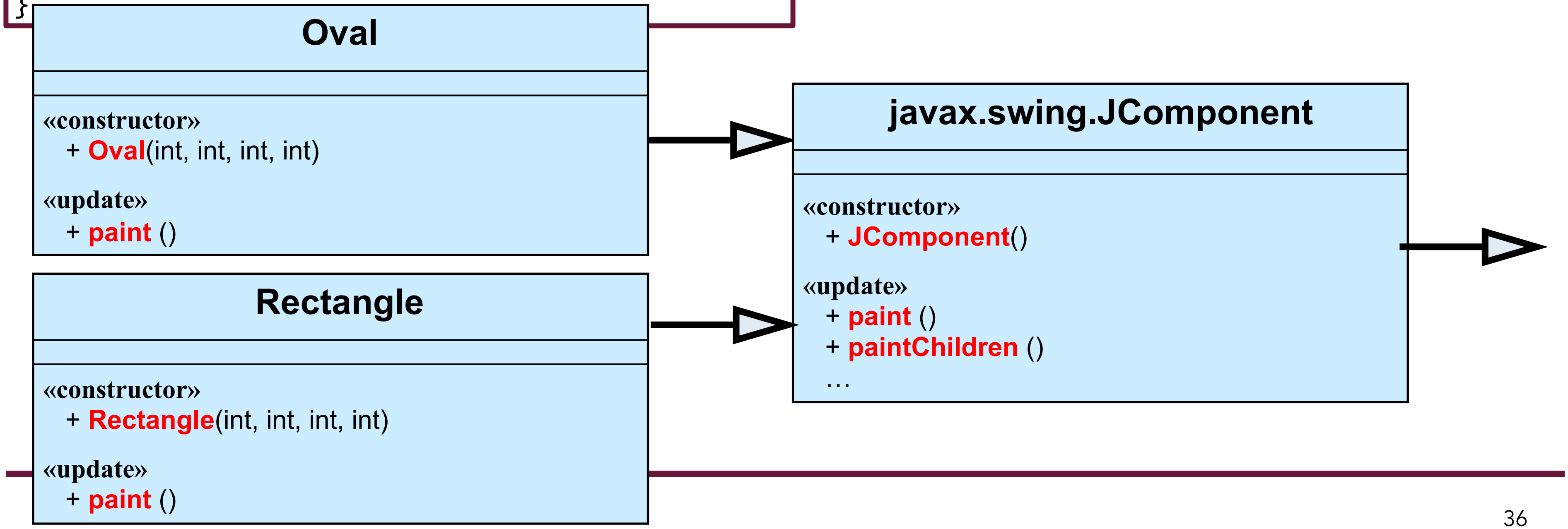
```
public class Example {
  public Example() {
    A aThing = new A();
    B bThing = new B();
    C cThing = new C();
    A reference;

    reference = aThing;
    reference.show();

    reference = bThing;
    reference.show();

    reference = cThing;
    reference.show();

    C otherRef;
    otherRef = aThing;
    otherRef.show();
  }
}
```

This is not ok since A is a superclass of C

Type Conformance Example: Window and JFrame

```
public class Window {
  private JFrame window;
  //...

  public void add( JComponent component ) {
    window.add( component, 0 );
    component.repaint();
  }

  public void remove( JComponent component ) {
    window.remove( component );
    window.repaint();
  }
}
```

Since Oval, Rectangle, Triangle, … objects are all subclasses of the JComponent then we can pass them into this common method.



Oval

«constructor»
+ Oval(int, int, int, int)

«update»
+ paint ()

Rectangle

«constructor»
+ Rectangle(int, int, int, int)

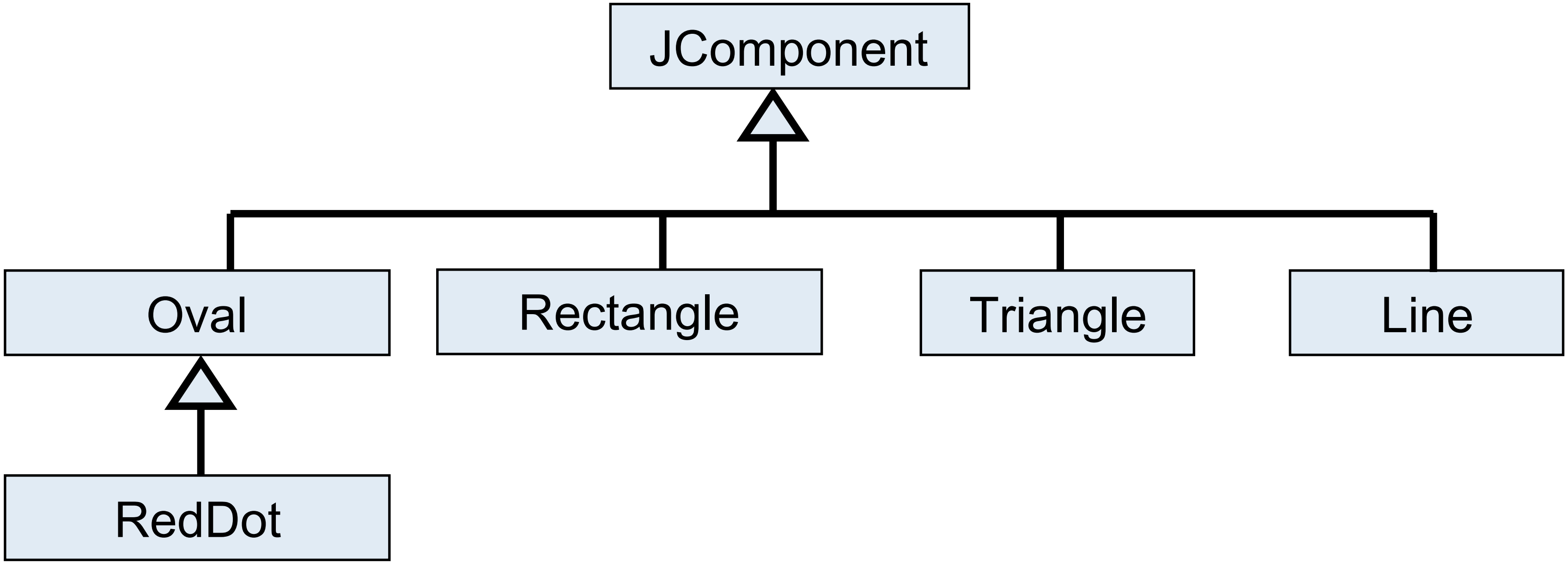
«update»
+ paint ()

javax.swing.JComponent

«constructor»
+ JComponent()

«update»
+ paint ()
+ paintChildren ()
…

Type Conformance Example: JComponents



```
public class Driver {
  private JComponent shapes[];

  public Driver( ) {
    shapes = new JComponent[3];
    shapes[0] = new Oval(20, 20, 10, 10);
    shapes[1] = new Rectangle(100, 100, 40, 50);
    shapes[2] = new Oval(20, 100, 10, 30);
  }
}
```

Type Conformance Example: JComponents

- **The instanceof operator allows us to determine the subclass of an object by comparison**

```
public class Driver {
  private JComponent shapes[];

  public Driver( ) {
    shapes = new JComponent[3];
    shapes[0] = new Oval(20, 20, 10, 10);
    shapes[1] = new Rectangle(100, 100, 40, 50);
    shapes[2] = new Oval(20, 100, 10, 30);

   for(int i = 0; i < shapes.length; ++i ) {
      if( shapes[i] instanceof Oval ) {
        System.out.println(i + " is an Oval");
      }
      else if( shapes[i] instanceof Rectangle ) {
        System.out.println(i + " is a Rectangle");
      }
      else {
        System.out.println(i + " is Unknown");
      }
    }
  }
}
```

```
object instanceof Class
```

```
0 is an Oval
1 is a Rectangle
2 is an Oval
```