Class Method

Expressions - Computing with Primitive Types

- For the primitive types int, double, and boolean, Java supports a notation for expressions that appeals to the one that we use in arithmetic and algebra courses.
- For example, we can write
  - 10 ∗ 12.50
  - width + height
  - Math.PI ∗ radius

Arthimetic and Relation Operators

Symbol	Parameter types	Result	Example	
+	numeric, numeric	numeric	x + 2	addition
-	numeric, numeric	numeric	x – 2	subtraction
*	numeric, numeric	numeric	x * 2	multiplication
/	numeric, numeric	numeric	x / 2	division

>	numeric	numeric	x > 2	greater than
>=	numeric, numeric	numeric	x >= 2	greater or equal
<	numeric, numeric	numeric	x < 2	less than
<=	numeric, numeric	numeric	x < 2	less or equal
==	numeric, numeric	numeric	x == 2	equal
!=	numeric, numeric	numeric	x != 2	not equal

Logic Operators

Symbol	Parameter types	Result	Example	
!	boolean	boolean	!(x < 0)	logical negation
&&	boolean, boolean	boolean	a && b	logical and
||	boolean, boolean	boolean	a || b	logical or

- Example

  (x != 0) && (x < 10) . . . determines whether a is not equal to x (int or double) and x is less than 10

Expressions - Method Calls

- A method is roughly like a function. Like a function, a method consumes data and produces data.
- However, a METHOD is associated with a class.
- Example:
  - To compute the length of the string in Java, we use the ***`Length`*** method from the ***`String`*** class like this:
    ```
    "hello world".length()
    ```
  - To concatenate "world" to the end of the argument "hello"
    ```
    String str = "hello";
    str.concat("world")
    ```
  - `Math.sqrt(10)` is square of 10

Method Calls

- When the method is called, it always receives at least one argument: an instance of the class with which the method is associated;
  - Because of that, a Java programmer does not speak of calling functions for some arguments, but instead speaks of **INVOKING** a method on an **instance or object**

- In general, a method call has this shape:

```
object.methodName(arg1, arg2, ...)
```

Design Class Method Steps

The design of methods follows the same design recipes

1. Problem analysis and data definitions
   - Specify pieces of information the method needs and output infomation
2. Purpose and contract (method signature)
   - The purpose statement is just a comment that describes the method's task in general terms.
   - The method signature is a specification of inputs and outputs, or **contract** as we used to call it.

Design Class Method Steps

3. Examples
   - the creation of examples that illustrate the purpose statement in a concrete manner
4. Method template
   - lists all parts of data available for the computation inside of the body of the method
5. Method definition
   - Implement method
6. Tests
   - to turn the examples into executable tests

Methods for Classes: Example

- Take a look at this revised version of our first problem
  . . . Design a method that computes the cost of selling bulk coffee at a specialty coffee seller from a receipt that includes the kind of coffee, the unit price, and the total amount (weight) sold. . .

Examples

1. 100 pounds of Hawaiian Kona at $15.95/pound

   $\rightarrow$ $1,595.00

2. 1,000 pounds of Ethiopian coffee at $8.00/pound

   $\rightarrow$ $8,000.00

3. 1,700 pounds of Colombian Supreme at $9.50/pound

   $\rightarrow$ 16,150.00

10

1. Problem analysis and data definitions

- Methods are a part of a class.
- Thus, if the **`CoffeeReceipt`** class already had a **`cost`** method, we could write:

```
new CoffeeReceipt("Kona", 15.95, 100).cost()
```

and expect this method call to produce 1595.0.

CoffeeReceipt
- String kind - double pricePerPound - double weight
??? cost(???)

1. Problem analysis and data definitions

- The only piece of information the method needs is the instance of the class **`CoffeeReceipt`** for which we are computing the selling cost.
- It will produce a double value that represents the selling cost.

2. Purpose and contract

- First we add a contract, a purpose statement, and a header for ***cost*** to the ***CoffeeReceipt*** class

```
// the bill for a CoffeeReceipt sale
class CoffeeReceipt {
   String kind;
   double pricePerPound; // in dollars per pound
   double weight; // in pounds
   CoffeeReceipt(String kind, double pricePerPound,
        double weight) { ... }

   // to compute the total cost of this coffee purchase
   // [in dollars]
   double cost() { ... }
}
```

a purpose statement

Contract is a METHOD SIGNATURE

Primary argument: `this`

- **`cost`** method is always invoked on some specific instance of **CoffeeReceipt**.
  - The instance is the primary argument to the method, and it has a standard name, **`this`**
- We can thus use **`this`** to refer to the instance of **`CoffeeReceipt`** and access to three pieces of data: the **`kind`**, the **`pricePerPound`**, and the **`weight`** in method body
  - Access field with: `object.field`
  - E.g: `this.kind`, `this.pricePerPound`, `this.weight`

3. Examples

- `new CoffeeReceipt("Kona", 15.95, 100).cost()`
  `// should produce 1595.0`
- `new CoffeeReceipt("Ethiopian", 8.0, 1000).cost()`
  `// should produce 8000.0`
- `new CoffeeReceipt("Colombian", 9.5, 20).cost()`
  `// should produce 190.0`

4. `cost` method template and result

```
// to compute the total cost of this coffee purchase
// [in   cents]
double cost() {
   ...this.kind...
   ...this.pricePerPound...
   ...this.weight...
}
```

The two relevant pieces are **`this.price`** and **`this.weight`**. If we multiply them, we get the result that we want:

```
// to compute the total cost of this coffee purchase
// [in   cents]
double cost() {
   return this.pricePerPound * this.weight;
}
```

5. `CoffeeReceipt` class and method

```
class CoffeeReceipt {
   String kind;
   double pricePerPound;
   double weight;

   CoffeeReceipt(String kind, double pricePerPound,
                 double weight) {
      this.kind = kind;
      this.pricePerPound = pricePerPound;
      this.weight = weight;
   }

   // to compute the total cost of this coffee purchase
   // [in dollars]
   double cost() {
      return this.pricePerPound * this.weight;
   }
}
```

6. Test `cost` method

```
import junit.framework.TestCase;
public class CoffeeReceiptTest extends TestCase {
   public void testContructor() {
      ...
   }

   public void testCost() {
      assertEquals(new CoffeeReceipt("Hawaiian Kona",
            15.95, 100).cost(), 1595.0);
      CoffeeReceipt c2 =
           new CoffeeReceipt("Ethiopian", 8.0, 1000);
      assertEquals(c2.cost(), 8000.0);
      CoffeeReceipt c3 =
           new CoffeeReceipt("Colombian Supreme ", 9.5, 1700);
      assertEquals(c3.cost(), 16150.0);
   }
}
```

Methods consume more data

Design method to such problems:

… The coffee shop owner may wish to find out whether a coffee sale involved a price over a certain amount …

CoffeeReceipt
- String kind - double pricePerPound - double weight
double cost() ??? priceOver(???)

Purpose statement and signature

- This method must consume two arguments:
  - given instance of coffee: **`this`**
  - a second argument, the **number of dollars** with which it is to compare the ***price*** of the sale's record.

inside of Coffee

```
// to determine whether this coffee's price is more
// than amount
boolean priceOver(double amount) { ... }
```

Examples

- `new CoffeeReceipt("Hawaiian Kona", 15.95, 100).priceOver(12) expected true`

- `new CoffeeReceipt("Ethiopian", 8.00, 1000).priceOver(12) expected false`

- `new CoffeeReceipt("Colombian Supreme ", 9.50, 1700).priceOver(12) expected false`

`priceOver` method template and result

```
// to determine whether this coffee's price
// is more than amount
boolean priceOver(int amount) {
   ... this.kind
   ... this.pricePerPound
   ... this.weight
   ... amount
}
```

The only relevant pieces of data in the template are *amt* and **this**.*price*:

```
// to determine whether this coffee's price
// is more than amount
boolean priceOver(int amount) {
   return this.pricePerPound > amount;
}
```

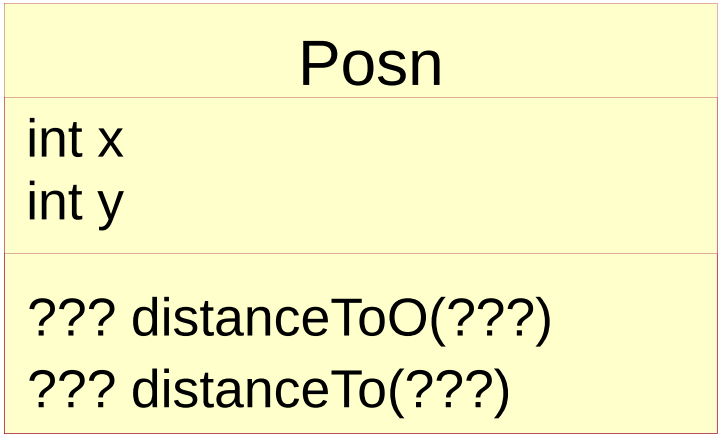
Test `priceOver` method

```
import junit.framework.TestCase;
public class CoffeeReceiptTest extends TestCase {
   ...

   public void testPriceOver() {
      CoffeeReceipt c1 =
           new CoffeeReceipt("Hawaiian Kona", 15.95, 100);
      CoffeeReceipt c2 =
           new CoffeeReceipt("Ethiopian", 8.00, 1000);
      CoffeeReceipt c3 =
           new CoffeeReceipt("Colombian Supreme ", 9.50, 1700);
      assertTrue(c1.priceOver(12));
      assertFalse(c2.priceOver(12));
      assertFalse(c3.priceOver(12));
   }
}
```

Posn example

- Suppose we wish to represent the pixels (colored dots) on our computer monitors.
  - A pixel is very much like a Cartesian point. It has an x coordinate, which tells us where the pixel is in the horizontal direction, and it has a y coordinate, which tells us where the pixel is located in the downwards vertical direction.
  - Given the two numbers, we can locate a pixel on the monitor
- Computes how far some pixel is from the origin
- Computes the distance between 2 pixels

Class diagram



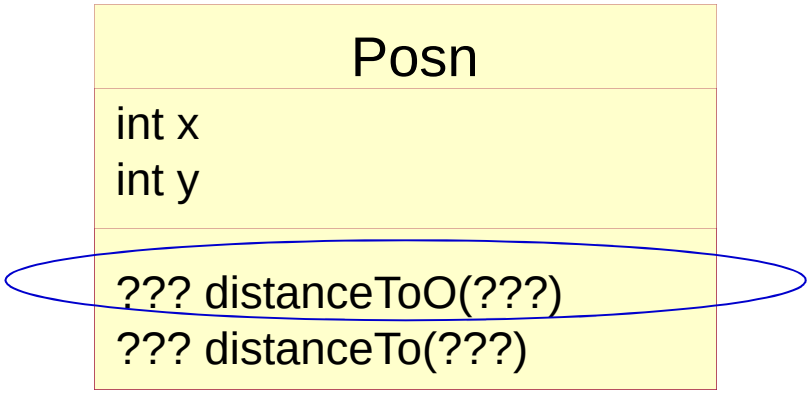
Define Class, Constructor, and Test

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

```
import junit.framework.*;

public class PosnTest extends TestCase {
   public void testConstrutor() {
      new Posn(5, 12);
      Posn aPosn1 = new Posn(6, 8);
      Posn aPosn2 = new Posn(3, 4);
   }
}
```

Computes

How far some pixel is from the origin



- Examples
  - Distance from A(5, 12) to O is 13
  - Distance from B(0, 3) to O is 9
  - Distance from A(3, 4) to O is 5

distanceToO method template

```
class Posn {
  int x;
  int y;

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

  // Computes how far this pixel is from the origin
  double distanceToO() {
    ...this.x...
    ...this.y...
  }
}
```

Add a contract,
a purpose statement

METHOD SIGNATURE

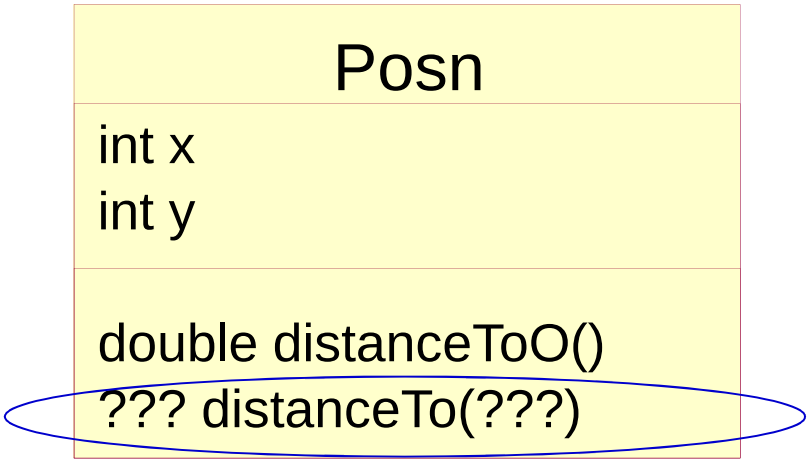
`distanceToO` method implementation

```
class Posn {
   int x;
   int y;

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

   double distanceToO() {
      return Math.sqrt(this.x * this.x + this.y * this.y);
   }
}
```

Test `distanceToO` method

```
import junit.framework.*;
public class PosnTest extends TestCase {
   ...

   public void testDistanceToO(){
      assertEquals(new Posn(5, 12).distanceToO(), 13.0, 0.001);
      Posn aPosn1 = new Posn(6, 8);
      assertEquals(aPosn1.distanceToO(), 10.0, 0.001);
      Posn aPosn2 = new Posn(3, 4);
      assertEquals(aPosn2.distanceToO(), 5.0, 0.001);
   }
}
```

Computes the distance between 2 pixels



- Example
  - Distance from A(6, 8) to B(3, 4) is 5
  - Distance from A(0, 3) to B(4, 0) is 5
  - Distance from A(1, 2) to B(5, 3) is

`distanceTo` method template

```
class Posn {
   int x;
   int y;
   ...
   // Computes how far this pixel is from the origin
   double distanceToO(){
      return Math.sqrt(this.x * this.x + this.y * this.y);
   }

   // Computes distance from this posn to another posn
   double distanceTo(Posn that) {
      ...this.x...this.y...
      ...that.x...that.y...
      ...this.distantoO()...                    )...
   }
}
```

Add a contract,
a purpose statement

METHOD SIGNATURE

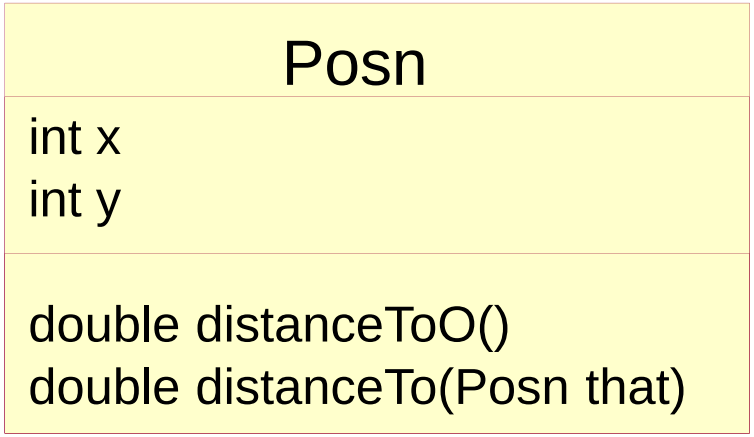
distanceTo method implement

```
class Posn {
   int x;
   int y;
   ...
   // Computes how far this pixel is from the origin
   double distanceToO(){
      return Math.sqrt(this.x * this.x + this.y * this.y);
   }

   // Computes distance from this posn to another posn
   double distanceTo(Posn that) {
      return Math.sqrt((that.x - this.x)*(that.x - this.x)
                     + (that.y - this.y)*(that.y - this.y));
  }
}
```

Test `distanceTo` method

```
import junit.framework.*;
public class PosnTest extends TestCase {
   ...

   public void testDistanceTo(){
      assertEquals(new Posn(6, 8).distanceTo(
                        new Posn(3, 4)), 5.0, 0.001);

      assertEquals(new Posn(0, 3).distanceTo(
                        new Posn(4, 0)), 5.0, 0.001);

      Posn aPosn1 = new Posn(1, 2);
      Posn aPosn2 = new Posn(5, 3);
      assertEquals(aPosn1.distanceTo(aPosn2), 4.1231, 0.001);

   }
}
```

Class diagram - Final

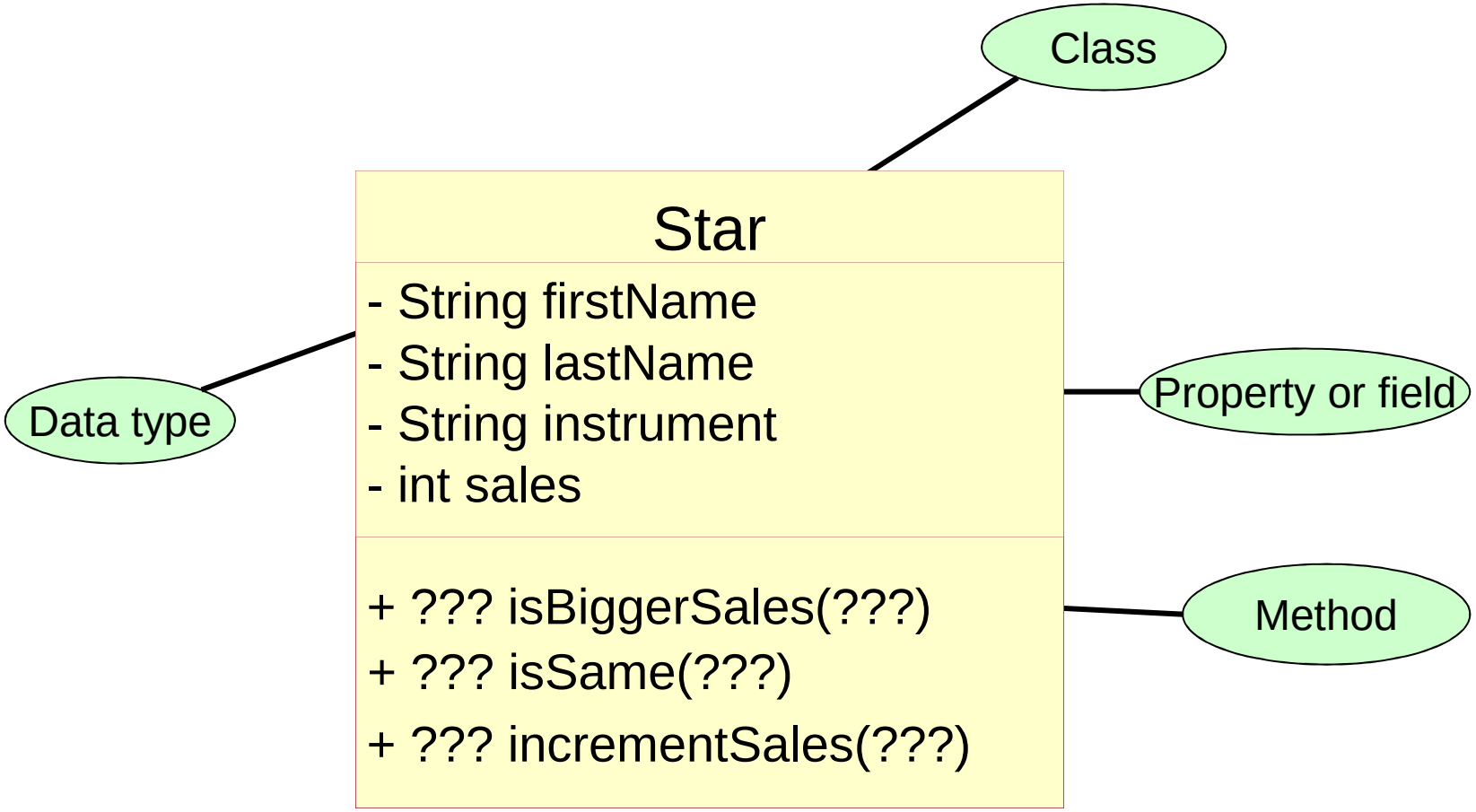


Object Compare

Star example

- Suppose we wish to represent a star information which has first name, last name, instrument he uses and his sales.
- Design methods:
  - Check whether one star's sales is greater than another star's sales.
  - Check whether one star is same another star.
  - Adds 20.000 to the star's sales.

Class Diagram



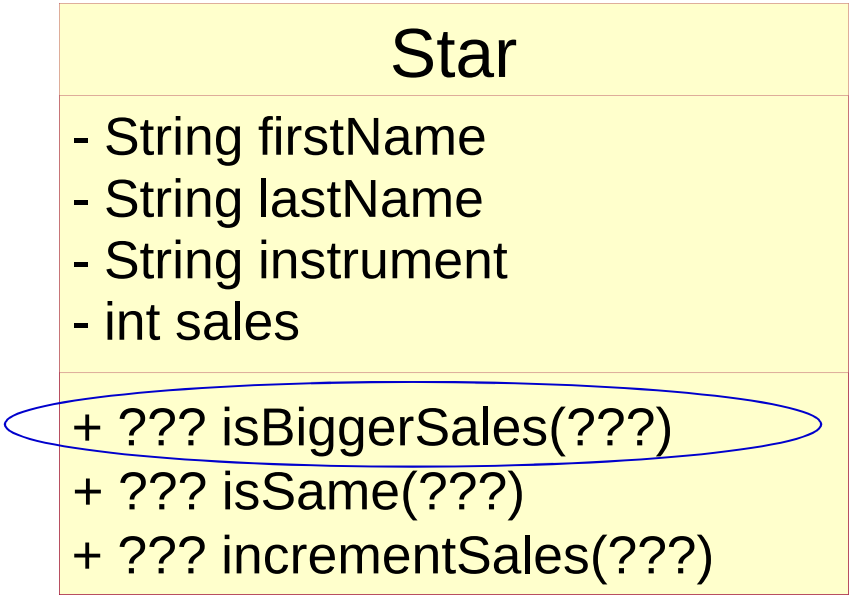
Define Class and Constructor

```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;

   // contructor
   public Star(String firstName, String lastName,
        String instrument, int sales) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.instrument = instrument;
      this.sales = sales;
   }
}
```

Test `Star` Constructor

```
import junit.framework.*;

public class TestStar extends TestCase {
   public void testConstructor() {
      new Star("Abba", "John", "vocals", 12200);
      Star aStar1 = new Star("Elton", "John", "guitar", 20000);
      Star aStar2 = new Star("Debie", "Gission", "organ", 15000);
   }
}
```

Check whether one star's sales is greater than another star's sales.



- Examples
  - new Star("Elton", "John", "guitar", 20000).isBiggerSales(new Star("Abba", "John", "vocals", 12200)) expected true

`isBiggerSales` method template

```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;
   ...

   // check whhether this star' sales is greater than
   // another star' sales
   public boolean isBiggerSales(Star other) {
      ...this.firstName...this.lastName...
      ...this.instrument...this.sales...
      ...other.firstName...other.lastName...
      ...other.instrument...other.sales...
   }
}
```

isBiggerSales method implement

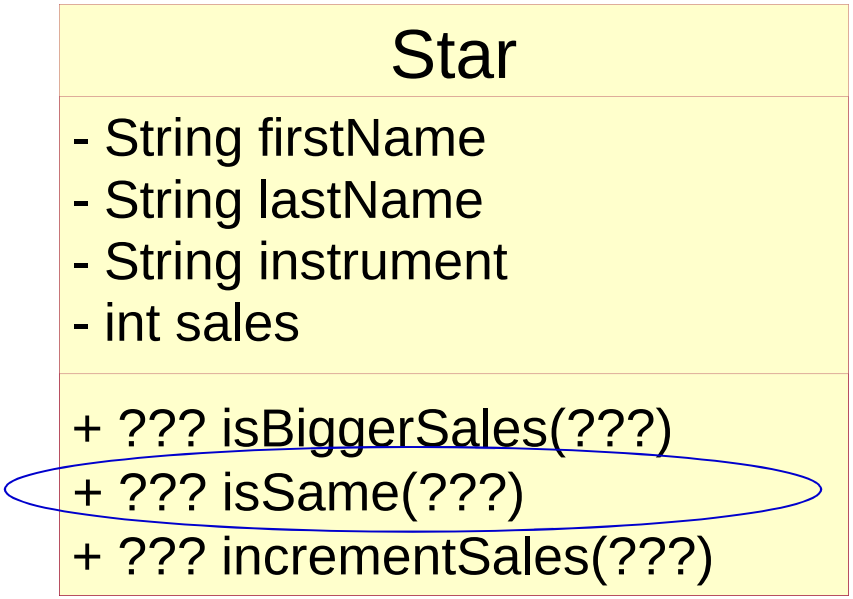
```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;
   ...

   // check whether this star is same another star
   public boolean isBiggerSales(Star other) {
      return (this.sales > other.sales);
   }
```

`isBiggerSales` method test

```
import junit.framework.TestCase;

public class StarTest extends TestCase {
   ...
   public void testIsBiggerSales () {
      Star aStar1 = new Star("Abba", "John", "vocals", 12200);
      assertTrue(new Star("Elton", "John", "guitar", 20000)
                     .isBiggerSales(aStar1));
      assertFalse(aStar1.isBiggerSales(
            new Star("Debie", "Gission", "organ", 15000)));
   }
}
```

Compare equals of 2 objects

- Check whether one star is same another star.



Design `isSame()` method

- `isSame` method template

```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;
   ...
   // check whhether this star is same another star
   public boolean isSame(Star other) {
      ...this.firstName...this.lastName...
      ...this.instrument...this.sales...
      ...this.isBigSales(...)
      ...other.firstName...other.lastName...
      ...other.instrument...other.sales...
      ...other.isBigSales(...)
  }
}
```

`isSame` method implement

```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;
   ...

   // check whether this star is same another star
   public boolean isSame(Star other) {
      return (this.firstName.equals(other.firstName)
            && this.lastName.equals(other.lastName)
            && this.instrument.equals(other.instrument)
            && this.sales == other.sales);
   }
}
```

`isSame` method test

```
import junit.framework.TestCase;
public class StarTest extends TestCase {
   ...
   public void testIsSame() {
      assertTrue(new Star("Abba", "John", "vocals", 12200)
         .isSame(new Star("Abba", "John", "vocals", 12200)));

      Star aStar1 = new Star("Elton", "John", "guitar", 20000);
      assertTrue(aStar1.isSame(
            new Star("Elton", "John", "guitar", 20000)));

      Star aStar2 = new Star("Debie", "Gission", "organ", 15000);
      Star aStar3 = new Star("Debie", "Gission", "organ", 15000);
      assertFalse(aStar1.isSame(aStar2));
      assertTrue(aStar2.isSame(aStar3));
   }
}
```

Other solution: `equals` method

- **A**: Why we do not use JUnit built-in assertEquals method?
- **Q**: Can override build-in `equals` method

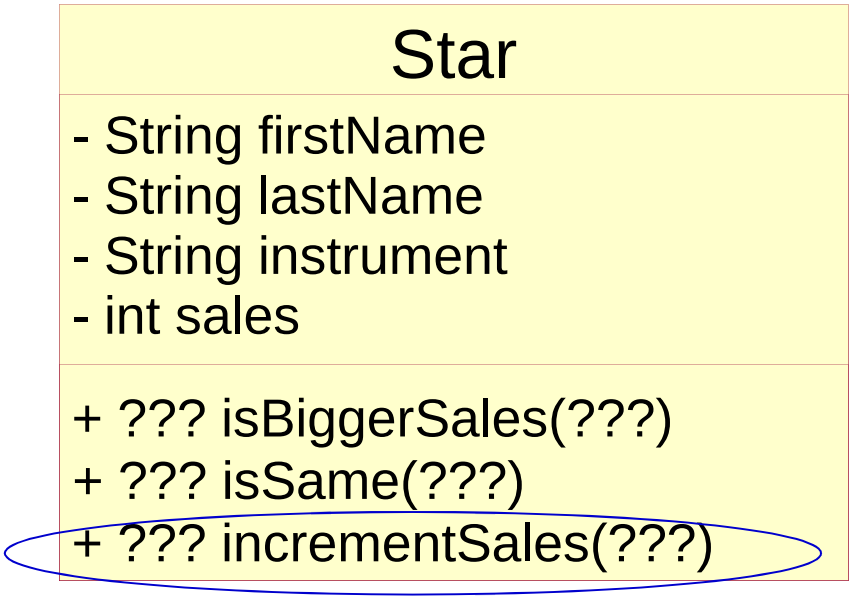
```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;
   ...
   public boolean equals(Object obj) {
      if (null == obj || !(obj instanceof Star))
         return false;
      else { Star that = (Star) obj;
         return this.firstName.equals(that.firstName)
             && this.lastName.equals(that.lastName)
             && this.instrument.equals(that.instrument)
             && this.sales.equals(that.sales);
      }
   }
}
```

`equals` method test

```
import junit.framework.TestCase;
public class StarTest extends TestCase {
   ...
   public void testEquals() {
      assertEquals(new Star("Abba", "John", "vocals", 12200),
                   new Star("Abba", "John", "vocals", 12200));

      Star aStar1 = new Star("Elton", "John", "guitar", 20000);
      assertEquals(aStar1,
                   new Star("Elton", "John", "guitar", 20000)));

      Star aStar2 = new Star("Debie", "Gission", "organ", 15000);
      Star aStar3 = new Star("Debie", "Gission", "organ", 15000);
      assertEquals(aStar2, aStar3);
   }
}
```

Change object state

- Adds 20.000 to the star's sales.



- 2 implements of `incrementSales` method
  - Immutable
  - Mutable

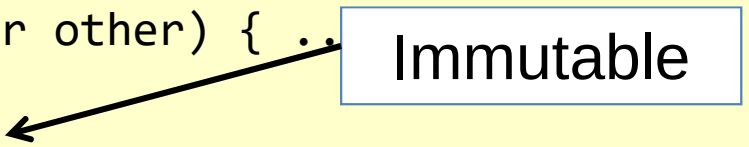
`incrementSales` method template

```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;
   ...

   // Adds 20.000 to the star's sales
   ??? incrementSales() {
      ...this.firstName...
      ...this.lastName...
      ...this.instrument...
      ...this.sales...
      ...this.isSame(...)...
      ...this.isBiggerSales(...)...
  }
}
```

`incrementSales` immutable

- creates a new star with a different sales.

```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;
   ...

   public boolean issame(Star other) { ... }
   public boolean isBiggerSales(Star other) { ...

   public Star incrementSales() {
      return new Star(this.firstName, this.lasttName,
                      this.instrument, this.sales + 20000);
   }
}
```



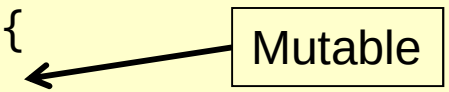
Test `incrementSales` immutable method

```
import junit.framework.*;
public class StarTest extends TestCase {
  ...
   public void testIncrementSales() {
      Star aStar1 = new Star("Abba", "John", "vocals", 12200);
      Star aStar2 = aStart1. incrementSales();
      assertTrue(aStart2.isSame(
                  new Star("Abba", "John", "vocals", 32200)));

      aStar1 = new Star("Elton", "John", "guitar", 20000);
      assertTrue(aStar1.incrementSales()
          .isSame(new Star("Elton", "John", "guitar", 40000)));

      assertTrue(new Star("Debie", "Gission", "organ", 15000)
          .incrementSales()
          .isSame(new Star("Debie", "Gission", "organ", 35000)));
   }
}
```

`mutableIncrementSales` method

- Change sales of `this` object

```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;
   ...
   public boolean issame(Star other) { ... }
   public boolean isBiggerSales(Star other) { ... }

   // Adds 20.000 to the star's sales
   public void mutableIncrementSales() {
      this.sales = this.sales + 20000
   }
```



Test mutableIncrementSales

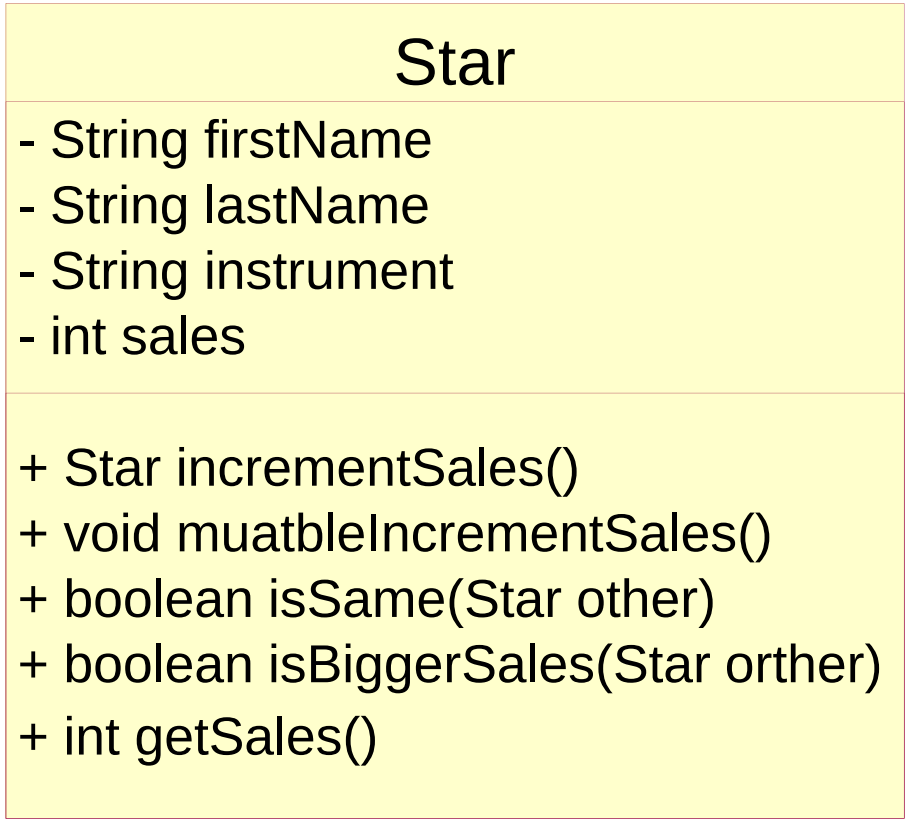
```
import junit.framework.*;

public class TestStar extends TestCase {
   ...

   public void testMutableIncrementSales (){
     Star aStar1 = new Star("Elton", "John", "guitar", 20000);
     Star aStar2 = new Star("Debie", "Gission", "organ", 15000);

     aStar1.mutableIncrementSales();
     assertEquals(40000, aStar1.getSales());

     aStar2. mutableIncrementSales();
     assertEquals(35000, aStar2.getSales());
   }
}
```

Discuss more: `getSales` method

- Q: Do we use "selector" **`this.sales`** outside **`Star`** class
- A: No
- Solution: **`getSales`** method

```
public class Star {
   private String firstName;
   private String lastName;
   private String instrument;
   private int sales;
   ...

   public int getSales() {
      return this.sales;
   }
}
```

Class diagram



Conditional Computations

Conditional Computations

- . . . Develop a method that computes the yearly interest for *certificates of deposit* (CD) for banks. The interest rate for a CD depends on the amount of deposited money. Currently, the bank pays 2% for amounts up to $5,000, 2.25% for amounts between $5,000 and $10,000, and 2.5% for everything beyond that. . . .

Define Class

```
public class CD {
    private String owner;
    private int amount; // cents

    public CD(String owner, int amount) {
        this.owner = owner;
        this.amount = amount;
    }
}
```

Example

- Translating the intervals from the problem analysis into tests yields three “interior” examples:
  - **new** *CD*("Kathy", 250000)*.interest*() **expect** 5000.0
  - **new** *CD*("Matthew", 510000)*.interest*() **expect** 11475.0
  - **new** *CD*("Shriram", 1100000)*.interest*() **expect** 27500.0

Conditional computation

- To express this kind of conditional computation, Java provides the so-called IF-STATEMENT, which can distinguish two possibilities:

```
if (condition) {
   statement1
}
```

```
if (condition) {
   statement1
}
else {
   statement2
}
```

`interest` method template

```
// compute the interest rate for this account
public double interest() {
   if (0 <= this.amount && this.amount < 500000) {
      ...this.owner...this.amount...
   }
   else  {
      if (500000 <= this.amount && this.amount < 1000000) {
          ...this.owner...this.amount...
      }
      else {
         ...this.owner...this.amount...
      }
   }
}
```

`interest()` method implement

```
// compute the interest rate for this account
public double interest() {
   if (0 <= this.amount && this.amount < 500000) {
      return 0.02 * this.amount;
   }
   else {
      if (500000 <= this.amount && this.amount < 1000000) {
         return 0.0225 * this.amount;
      }
      else {
         return 0.025 * this.amount;
      }
   }
}
```

`interest()` full implement

```
// compute the interest rate for this account
public double interest() {
   if (this.amount < 0) {
      return 0;
   }
   else {
      if (this.amount < 500000) {
         return 0.02 * this.amount;
      }
      else {
         if (this.amount < 1000000) {
            return 0.0225 * this.amount;
         }
         else {
            return 0.025 * this.amount;
         }
      }
   }
}
```

`interest()` different implement

```
// compute the interest rate for this account
public double interest() {
   if (this.amount < 0) {
      return 0;
   }
   if (this.amount < 500000) {
      return 0.02 * this.amount;
   }
   if (this.amount < 1000000) {
      return 0.0225 * this.amount;
   }
   return 0.025 * this.amount;
}
```