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
%	integer, integer	integer	x % y	modulo

>	numeric, numeric	boolean	x > 2	greater than
>=	numeric, numeric	boolean	x >= 2	greater or equal
<	numeric, numeric	boolean	x < 2	less than
<=	numeric, numeric	boolean	x < 2	less or equal
==	numeric, numeric	boolean	x == 2	equal
!=	numeric, numeric	boolean	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;
- 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

Coffee Seller 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
     → $1,595.00
  2) 1,000 pounds of Ethiopian coffee at $8.00/pound
     → $8,000.00
  3) 1,700 pounds of Colombian Supreme at $9.50/pound
     → 16,150.00

1. Problem analysis and data definitions

```java
class Coffee {
   String kind;
   double price;
   double weight;
   Coffee(String kind, double price,
          double weight) {
      this.kind = kind;
      this.price = price;
      this.weight = weight;
   }
}
```

Coffee
- String kind - double price - double weight

```java
import junit.framework.*;
public class CoffeeTest extends TestCase {
    public void testConstructor() {
        new Coffee("Hawaiian Kona", 15.95, 100);
        new Coffee("Ethiopian", 8.0, 1000);
        new Coffee("Colombian Supreme ", 9.5, 1700);
    }
}
```

1. Problem analysis and data definitions

- Methods are a part of a class.
- Thus, if the **`Coffee`** class already had a **`cost`** method, we could write:
  `new Coffee("Kona", 15.95, 100).cost()`
  and expect this method call to produce 1595.0.
- The only piece of information the method needs is ***the instance of the class*** **`Coffee`** for which we are computing the selling cost.
- It will produce a double value that represents the selling cost.

Coffee
- String kind - double price - double weight
??? cost(???)

2. Purpose and contract

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

```
// the bill for a Coffee sale
class Coffee {
   String kind;
   double price; // in dollars per pound
   double weight; // in pounds
   Coffee(String kind, double price, 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 **Coffee**.
  - 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 `Coffee` and access to three pieces of data: the `kind`, the `price`, and the `weight` in method body
  - Access field with: `object.field`
  - E.g: `this.kind`, `this.price`, `this.weight`

3. Examples

- `new Coffee("Hawainian Kona", 15.95, 100).cost()`
  // should produce 1595.0
- `new Coffee("Ethiopian", 8.0, 1000).cost()`
  // should produce 8000.0
- `new Coffee("Colombian", 9.5, 1700).cost()`
  // should produce 16150.0

4. `cost` method template and result

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

5. Coffee class and method

```
class Coffee {
   String kind;
   double price;
   double weight;

   Coffee(String kind, double price, double weight) {
      this.kind = kind;
      this.price = price;
      this.weight = weight;
   }

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

6. Test cost method

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

   public void testCost() {
      assertEquals(
        new Coffee("Hawaiian Kona", 15.95, 100).cost(), 1595.0);
      Coffee c2 = new Coffee("Ethiopian", 8.0, 1000);
      assertEquals(c2.cost(), 8000.0);
      Coffee c3 = new Coffee("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 …

Coffee
- String kind - double price - 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.

<u>inside of Coffee</u>

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

Examples

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

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

- `new Coffee("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(double amount) {
   ... this.kind
   ... this.price
   ... this.weight
   ... amount
}
```

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

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

Test priceOver method

```java
import junit.framework.TestCase;
public class CoffeeTest extends TestCase {
   ...

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

Catesian Point 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 and Test

CartPt
int x int y

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

```java
import junit.framework.*;
public class CartPtTest extends TestCase {
   public void testConstrutor() {
      new CartPt(5, 12);
      CartPt aCartPt1 = new CartPt(0, 3);
      CartPt aCartPt2 = new CartPt(3, 4);
   }
}
```

Computes

How far some pixel is from the origin

CartPt
int x int y
??? distanceToO(???) ??? distanceTo(???)

`distanceToO` method signature

inside of CartPt

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

- Examples
  - `new CartPt(5, 12).distanceToO()` should be 13.0
  - `new CartPt(0, 3).distanceToO()` should be 3.0
  - `new CartPt(4, 7).distanceToO()` should be 8.062

distanceToO method template

```
class CartPt {
  int x;
  int y;

  CartPt(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

distanceTo0 method implementation

```java
class CartPt {
    int x;
    int y;

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

    // Computes how far this pixel is from the origin
    double distanceTo0() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
}
```

Test distanceToO method

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

   public void testDistanceToO() {
      assertEquals(new CartPt(5, 12).distanceToO(), 13.0,
0.001);
      CartPt aCartPt1 = new CartPt(0, 3);
      assertEquals(aCartPt1.distanceToO(), 3.0, 0.001);
      CartPt aCartPt2 = new CartPt(4, 7);
      assertEquals(aCartPt2.distanceToO(), 8.062, 0.001);
   }
}
```

Computes the distance between 2 pixels

CartPt
int x int y
double distanceToO() ??? distanceTo(???)

distanceTo Method Signature

inside of CartPt

```
// Computes distance from this CartPt to another
CartPt
double distanceTo(CartPt that) { ... }
```

- Examples
  - `new CartPt(6, 8).distanceTo(new CartPt(3, 4))`
    should be 5.0
  - `new CartPt(0, 3).distanceToO(new CartPt(4, 0))` should be 5.0
  - `new CartPt(1, 2).distanceToO(new CartPt(5, 3))`
    should be 4.123

distanceTo method template

```
class CartPt {
   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 CartPt to another CartPt
   double distanceTo(CartPt that) {
      ...this.x...this.y...
      ...that.x...that.y...
      ...this.distantoO()...that.distanceToO()...
   }
}
```

Add a contract, a purpose statement
METHOD SIGNATURE

distanceTo method implement

```
class CartPt {
   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 CartPt to another CartPt
   double distanceTo(CartPt 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 CartPtTest extends TestCase {
   ...

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

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

      CartPt aCartPt1 = new CartPt(1, 2);
      CartPt aCartPt2 = new CartPt(5, 3);
      assertEquals(aCartPt1.distanceTo(aCartPt2), 4.123, 0.001);

  }
}
```

Class diagram - Final

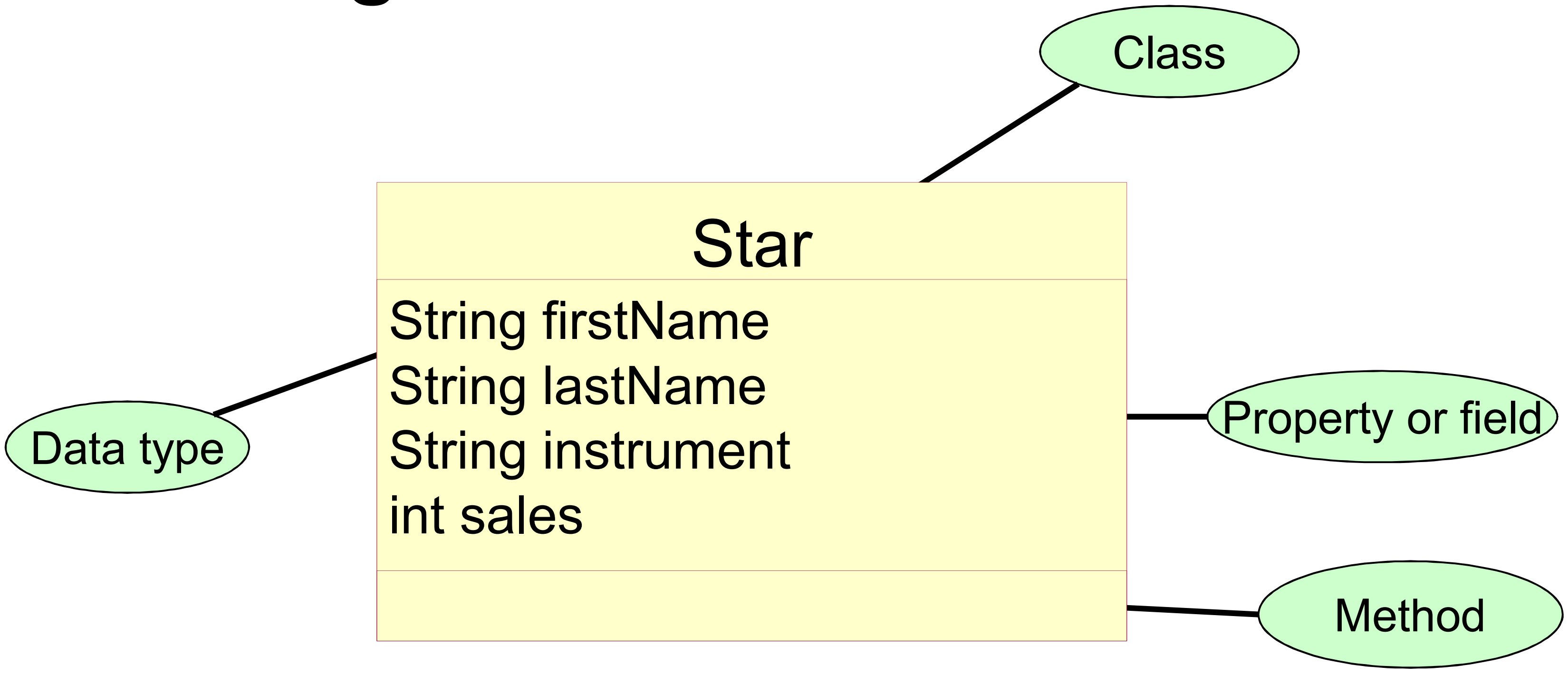
CartPt
int x int y
double distanceToO() double distanceTo(CartPt that)

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.

Class Diagram



Define Class and Constructor

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

   // contructor
   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.*;

class TestStar extends TestCase {
   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.

Star
String firstName String lastName String instrument int sales
??? biggerSales(???)

- Examples

```
new Star("Elton", "John", "guitar", 20000)
.biggerSales(new Star("Abba", "John", "vocals", 12200))
```

  expected true

biggerSales method template

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

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

biggerSales method implement

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

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

biggerSales method test

```
import junit.framework.TestCase;

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

Compare equals of 2 objects

- Check whether one star is same another star.

Star
String firstName String lastName String instrument int sales
boolean biggerSales(Star other) ??? same(???)

same() method template

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

   // check whhether this star is same another star
   boolean same(Star other) {
      ...this.firstName...this.lastName...
      ...this.instrument...this.sales...
      ...this.isBigSales(...)
      ...other.firstName...other.lastName...
      ...other.instrument...other.sales...
      ...other.isBigSales(...)
  }
}
```

same method implement

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

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

`same` method test

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

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

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

Other solution: `equals` method

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

```
class Star {
   String firstName;
   String lastName;
   String instrument;
   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 == 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);
   }
}
```

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

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

    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
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
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
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

```java
// compute the interest rate for this account
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;
}
```

Mutable and Inmutable methods

Star example

- Suppose we wish to represent a star information which has first name, last name, instrument he uses and his sales.
- Design methods:
  - Adds 20.000 to the star's sales.

Adds 20.000 to the star's sales.

Star
String firstName String lastName String instrument int sales
??? biggerSales(???) ??? same(???) ??? incrementSales(???)

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

incrementSales method template

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

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

Don't change object state

- `incrementSales` immutable: creates a new star with a different sales.

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

   boolean same(Star other) { ... }
   boolean biggerSales(Star other) { ... }
   // Create another star with 20.000 add to this star's sales
   Star incrementSales() {
      return new Star(this.firstName, this.lasttName,
                      this.instrument, this.sales + 20000);
   }
}
```

Immutable

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.same(
                  new Star("Abba", "John", "vocals", 32200)));

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

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

Change object state

- `mutableIncrementSales` method: Change sales of `this` object

```
class Star {
   String firstName;
   String lastName;
   String instrument;
   int sales;
   ...
   boolean same(Star other) { ... }
   boolean biggerSales(Star other) { ... }

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

Mutable

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

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

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

Class diagram

Star
String firstName String lastName String instrument int sales
Star incrementSales() void muatbleIncrementSales() boolean same(Star other) boolean biggerSales(Star orther) int getSales()