COSC 111:
Computer Programming I

Dr. Bowen Hui

University of British Columbia
Okanagan

Administration

- Midterm 2
  - To return next class
- Assignment 3
  - Available on website
  - Due next Thursday
- Today
  - Static modifier
  - Overloading
  - Pass by value vs. pass by reference

The static modifier

- Can declare methods and variables to be static
- Associates them to the class
  - *Not* to the object of the class
- E.g.: Math.sqrt(4)
  sqrt() is a static method in Math class
- E.g.: Board.BOARD_WIDTH
  BOARD_WIDTH is a static variable in Board class
- Must be called through class name

Static Variables

- Also called class variables
- Normally, each object has its own data space
- If var is static, only one copy of var exists
- Memory space for a static var is created when the class is first referenced
- All objects instantiated from the class share its static vars

Example

```
public class Example
{
    public static String hello = “hi”;
}
public class TestExample
{
    public static void main( String[] args )
    {
        // no need to create any object
        System.out.println( Example.hello );
    }
}
```

Example 2

```
public class Example
{
    public static String hello = "hi";
    private String name;

    public Example( String n ) {    name = n;    }

    public void changeStr() {    hello += " " + name;  }
}
```

Example 2 cont.

```
public class TestExample
{
    public static void main( String[] args )
    {
        System.out.println( Example.hello );
        Example one = new Example( "casper" );
        Example two = new Example( "bitzy" );
        System.out.println( Example.hello );
        one.changeStr();
        System.out.println( Example.hello );
    }
}
```

Example 2 cont.

```
public class TestExample
{
    public static void main( String[] args )
    {
        System.out.println( Example.hello );
        Example one = new Example( "casper" );
        Example two = new Example( "bitzy" );
        System.out.println( Example.hello );
        one.changeStr();
        System.out.println( Example.hello );
    }
}
```

this line prints:
hi

Example 2 cont.

```
public class TestExample
{
    public static void main( String[] args )
    {
        System.out.println( Example.hello );
        Example one = new Example( "casper" );
        Example two = new Example( "bitzy" );
        System.out.println( Example.hello );
        one.changeStr();
        System.out.println( Example.hello );
    }
}
```

this line prints:
hi

this line prints:
hi

Example 2 cont.

```
public class TestExample
{
    public static void main( String[] args )
    {
        System.out.println( Example.hello );
        Example one = new Example( "casper" );
        Example two = new Example( "bitzy" );
        System.out.println( Example.hello );
        one.changeStr();
        System.out.println( Example.hello );
    }
}
```

this line prints:
hi

this line prints:
hi

this line prints:
hi casper

When to use Static variables

- Determined based on program design
- Most common use: constants
  - private final static TYPE VAR = …
  - public final static TYPE VAR = …

When to use Static variables

- Determined based on program design
- Most common use: constants
  - private final static TYPE VAR = …
  - public final static TYPE VAR = …
- Any other common resource shared among objects of a class
  - E.g., keeping track of how many objects of the class have been created
  - E.g., all objects write to same file

Static Methods

- Also called class methods
- Static methods can be called through the class name
- Static methods cannot reference non-static class attributes
  - Non-static class attributes only exist with objects
  - Static methods don't require objects to be created

The main() method

- None of the test classes ever require objects
- main() indicates the starting point of a program
- Recall:

```
public class TestExample
{
    public static void main( String[] args )
    {
        System.out.println( Example.hello );
    }
}
```

When to use Static Methods

- Determined based on program design
- Most common use: utility methods
  - Methods you use often across classes and programs
  - E.g., calculations, conversions

When to use Static Methods

- Determined based on program design
- Most common use: utility methods
  - Methods you use often across classes and programs
  - E.g., calculations, conversions
- If the method does not depend on object creation
- If the method definition does not change (next semester: some method definitions can be overridden)

Method Overloading

- Method overloading is the process of giving a single method name to multiple definitions in a class
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The signature of each overloaded method must be unique
  - Signature includes the number, type, and order of the input parameters
  - This is a part of the method header (what's missing?)

Example

- Within the same class, we have:
  float calc(int x) { return x + 0.375; }
  float calc(int x, float y) { return x * y; }
- Which method are we calling if:
  - result = calc(25, 4.32);

Example

- Within the same class, we have:
  float calc(int x) { return x + 0.375; }
  float calc(int x, float y) { return x * y; }
- Which method are we calling if:
  - result = calc(25, 4.32);
  - result = calc(3);

The println() method

- println() is overloaded
- Consider the following signatures:
  - println(String a)
  - println(int i)
  - println(double d)
  - …
- Each example calls a different println() definition
  - System.out.println(“Cost is:”);
  - System.out.println(total);

Unique Signatures

- Signatures pertain to input parameters
- Can we have the following methods in the same class?

```
float calc( int x )  {  return x + 0.375; }
int calc( int x )  {  return x * 2; }
```

Unique Signatures

- Signatures pertain to input parameters
- Can we have the following methods in the same class?

```
float calc( int x )  {  return x + 0.375; }
float calc( float x )  {  return x * 2; }
```

When to use overloading?

- Common use to overload constructor methods
  - Provides multiple ways to initialize a new object

When to use overloading?

- Common use to overload constructor methods
  - Provides multiple ways to initialize a new object
- When input parameters differ in types

```
float calc( int x )  {  … }
float calc( float x )  {  … }
```

When to use overloading?

- Common use to overload constructor methods
  - Provides multiple ways to initialize a new object
- When input parameters differ in types
  float calc(int x) { … }
  float calc(float x) { … }
- When the input parameters vary in type and number
  float calc(int x) { … }
  float calc(float x, int y) { … }

Pass by Value vs. Pass by Reference

- Matters when we pass different info as parameters to a method call
- Changes made inside the method …
  - Stay changed after the method call (pass by reference)
  - Do not change after method call (pass by value)

Example

```
public class Num
{
  private int value;

  public Num( int val ) { value = val; }
  public void setValue( int newValue ) { value = newValue; }
  public String toString()
  {
    String str = "";
    str += value;
    return str;
  }
}
```

Example (cont.)

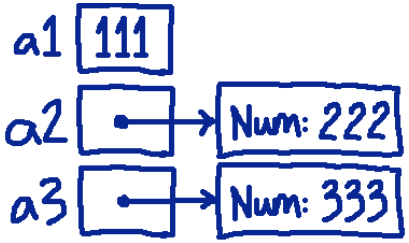
```
public class ParameterModifier
{
  public void changeValue( int f1, Num f2, Num f3 )
  {
    System.out.println( "before changing values: ");
    System.out.println( "f1\t f2\t f3" );
    System.out.println( f1 + "\t " + f2 + "\t " + f3 + "\n" );

    f1 = 999;
    f2.setValue( 888 );
    f3 = new Num( 777 );

    System.out.println( "after changing values: ");
    System.out.println( "f1\t f2\t f3" );
    System.out.println( f1 + "\t " + f2 + "\t " + f3 + "\n" );
  }
}
```

Example (cont.)

```
public class TestParams
{
  public static void main( String[] args )
  {
    ParameterModifier modifier = new ParameterModifier();

    int a1 = 111;
    Num a2 = new Num( 222 );
    Num a3 = new Num( 333 );

    System.out.println( "before changing values: ");
    System.out.println( "a1\t a2\t a3" );
    System.out.println( a1 + "\t " + a2 + "\t " + a3 + "\n" );

    modifier.changeValue( a1, a2, a3 );

    System.out.println( "after changing values: ");
    System.out.println( "a1\t a2\t a3" );
    System.out.println( a1 + "\t " + a2 + "\t " + a3 + "\n" );
  }
}
```

Visually

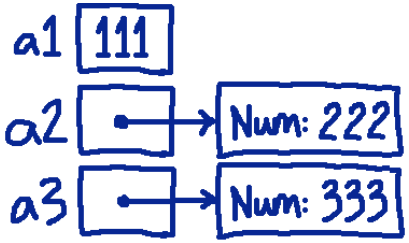
1. Before method:



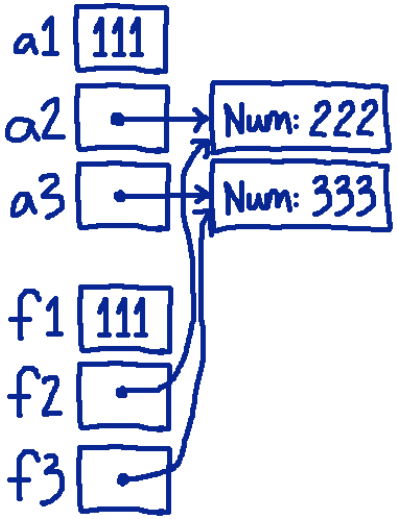

```
public class TestParams
{
  public static void main( String[] args )
  {
    ParameterModifier modifier = new ParameterModifier();

    int a1 = 111;
    Num a2 = new Num( 222 );
    Num a3 = new Num( 333 );

    System.out.println( "before changing values: ");
    System.out.println( "a1\t a2\t a3" );
    System.out.println( a1 + "\t " + a2 + "\t " + a3 + "\n" );

    modifier.changeValue( a1, a2, a3 );
```

Visually

1. Before method:



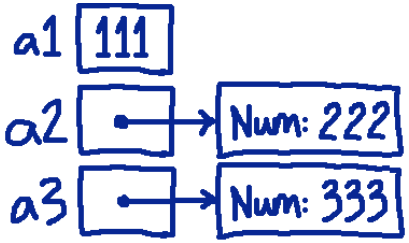
2. Inside method (before):



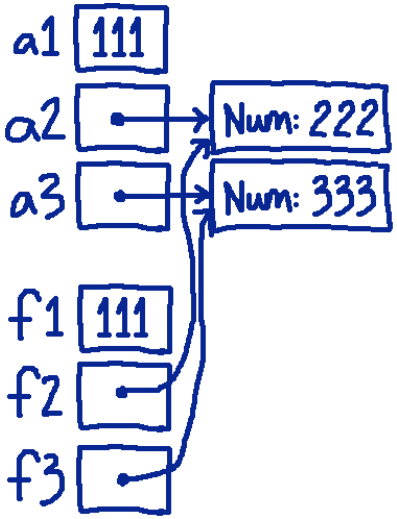

```
public class ParameterModifier
{
  public void changeValue( int f1, Num f2, Num f3 )
  {
    System.out.println( "before changing values: ");
    System.out.println( "f1\t f2\t f3" );
    System.out.println( f1 + "\t " + f2 + "\t " + f3 + "\n" );
```

Visually

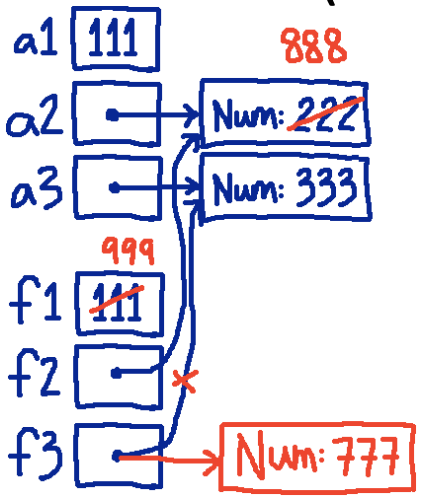
1. Before method:



2. Inside method (before):



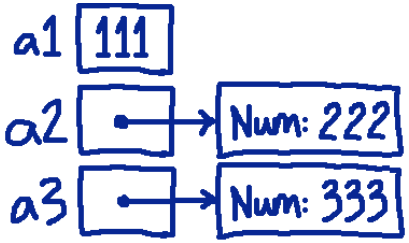
3. Inside method (after):



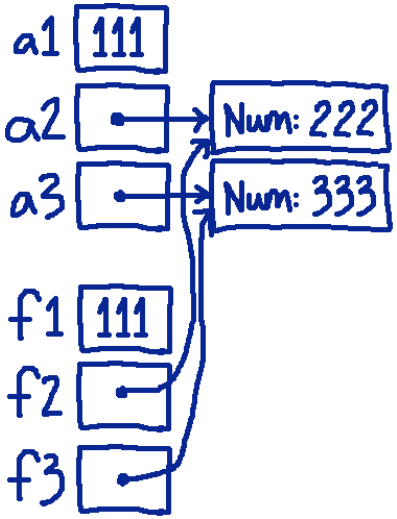

```
f1 = 999;
f2.setValue( 888 );
f3 = new Num( 777 );

System.out.println( "after changing values: ");
System.out.println( "f1\t f2\t f3" );
System.out.println( f1 + "\t " + f2 + "\t " + f3 + "\n" );
```

Visually

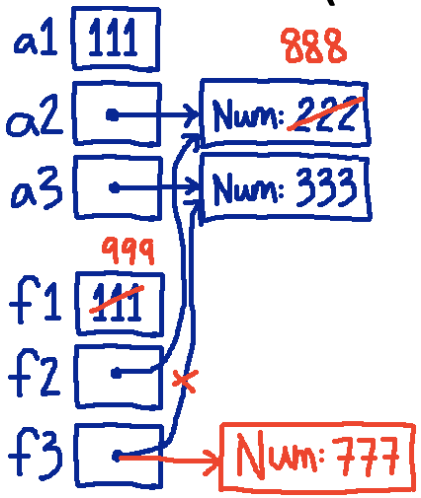
1. Before method:



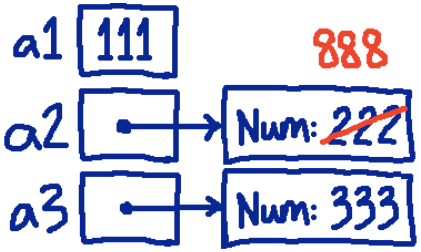
2. Inside method (before):



3. Inside method (after):



4. After method:



Example (cont.)

- Output:

```
before changing values:
a1        a2        a3
111       222       333

before changing values:
f1        f2        f3
111       222       333

after changing values:
f1        f2        f3
999       888       777

after changing values:
a1        a2        a3
111       888       333
```

- Notes:
  - inside method: f1,f2,f3 change as expected
  - outside method:
    - a1 doesn't change
    - a2 stays changed
    - a3 is a new object

Rules for Pass by Value

- Variable data is duplicated and used inside that method
- All method variables will be destroyed when exit method
- Original variable will be unchanged
- In Java, all primitive types follow this rule
  - E.g.: int, char, double, float, etc. (8 possible)

Rules for Pass by Reference

- Object is passed into method via its address
  - Its pointer, its reference (multiple names)
- All reference variables will be destroyed when exit method
- Original object variable will also be changed because addresses are used
- In Java, all objects follow this rule
  - E.g.: String, arrays, custom class objects you create

Applying these concepts in A3

- Q1 – makes use of overloading
- Q2 – makes use of pass by reference/value
- Q3 – makes use of static variables
- Generally, more practice on:
  - Loops
  - Arrays
  - Passing objects as parameters