Classes (Part 4)

SE 206

Objects Life Cycle

- Stages
  - creation
  - use
  - cleanup

Creation of an object

- Class Declaration (providing name and definition for the object)
- instantiation (setting aside memory for the object)
- optional initialization (providing initial values for the object via constructors)

Use of an object

- We activate the behavior of an object by *invoking one of its methods* (sending it a message).
- When an object receives a message (has one of its methods invoked), it either *performs an action*, or *modifies its state*, or both.

Cleanup

- What happens to all the objects that are instantiated?
  - When an object is no longer needed, we simply forget it. Eventually, the garbage collector may (or may not) come by and pick it up for recycling

Cleanup Approach

- **The good news**
  - if things work as planned, the Java programmer never needs to worry about returning memory to the operating system. This is taken care of automatically by a feature of Java known as the *garbage* collector.
- **The bad news**
  - Java does not support anything like a *destructor* that is guaranteed to be called whenever the object goes out of scope or is no longer needed. Therefore, other than returning allocated memory, it is the responsibility of the programmer to explicitly perform any other required cleanup at the appropriate point in time.
  - Other kinds of cleanup could involve closing files, disconnecting from open telephone lines, etc.

Garbage Collection

- The sole purpose of *garbage collection* is to reclaim memory occupied by objects that are no longer needed.
- Eligibility for garbage collection
  - An object becomes *eligible* for *garbage collection* when there are no more references to that object. You can make an object *eligible* for garbage collection by setting all references to that object to **null**, or allowing them to go out of scope.

No Guarantees

- However, just because an object is *eligible* for *garbage collection* doesn't mean that it will be reclaimed.
- The *garbage collector* runs in a low-priority thread, and presumably is designed to create minimal interference with the other threads of the program. Therefore, the garbage collector may not run unless a memory shortage is detected. And when it does run, it runs asynchronously relative to the other threads in the program.

Finalize Method

- Before the *garbage collector* reclaims the memory occupied by an object, it calls the object's **finalize()** method.
- The **finalize()** method is a member of the **Object** class. Since all classes inherit from the **Object** class, your classes also contain the default **finalize()** method. This gives you an opportunity to execute your special cleanup code on each object before the memory is reclaimed
- In order to make use of the **finalize()** method, you must *override* it, providing the code that you want to have executed before the memory is reclaimed.

When do I use the finalize() method

- Is the cleanup timing critical?
  - If you simply need to do cleanup work on an object sometime before the program terminates, (and you have specified finalization on exit) you can ALMOST depend on your overridden **finalize()** method being executed sometime before the program terminates.
  - If you need cleanup work to be performed earlier (such as disconnecting an open long-distance telephone call), you must explicitly call methods to do cleanup at the appropriate point in time and not depend on finalization to get the job done.
- If you use the finalize() method, make sure that you call the super.finalize() method at the end of it.

Static vs. non-static

Methods

- Instance (or member) method
  - Operates on a object (i.e., and *instance* of the class)

```
String s = new String("Help every cow reach its "
      + "potential!");
int n = s.length();
```

Instance method

- Class (i.e. static) method
  - Service provided by a class and it is not associated with a particular object

```
String t = String.valueOf(n);
```

Class method

Variables

- Instance variable and instance constants
  - Attribute of a particular object
  - Usually a variable

```
Point p = new Point(5, 5);
int px = p.x;
```

Instance variable

- Class variables and constants
  - Collective information that is not specific to individual objects of the class
  - Usually a constant

```
Color favoriteColor = Color.MAGENTA;
double favoriteNumber = Math.PI - Math.E;
```

Class constants

static and non-static rules

- Member/instance (i.e. non-static) fields and methods can ONLY be accessed by the object name

- Class (i.e. static) fields and methods can be accessed by Either the class name or the object name

Static vs. non-static (Read yourself)

- Consider the following code:

```
public class Staticness {

    private int a = 0;
    private static int b = 0;

    public void increment() {
        a++;
        b++;
    }

    public String toString() {
        return "(a=" + a + ",b=" + b + ")";
    }

}
```

Static vs. non-static (Read yourself)

- And the code to run it:

```
public class StaticTest {

    public static void main (String[] args) {
        Staticness s = new Staticness();
        Staticness t = new Staticness();

        s.increment();
        t.increment();
        t.increment();

        System.out.println (s);
        System.out.println (t);
    }
}
```

Static vs. non-static (Read yourself)

- Execution of the code…

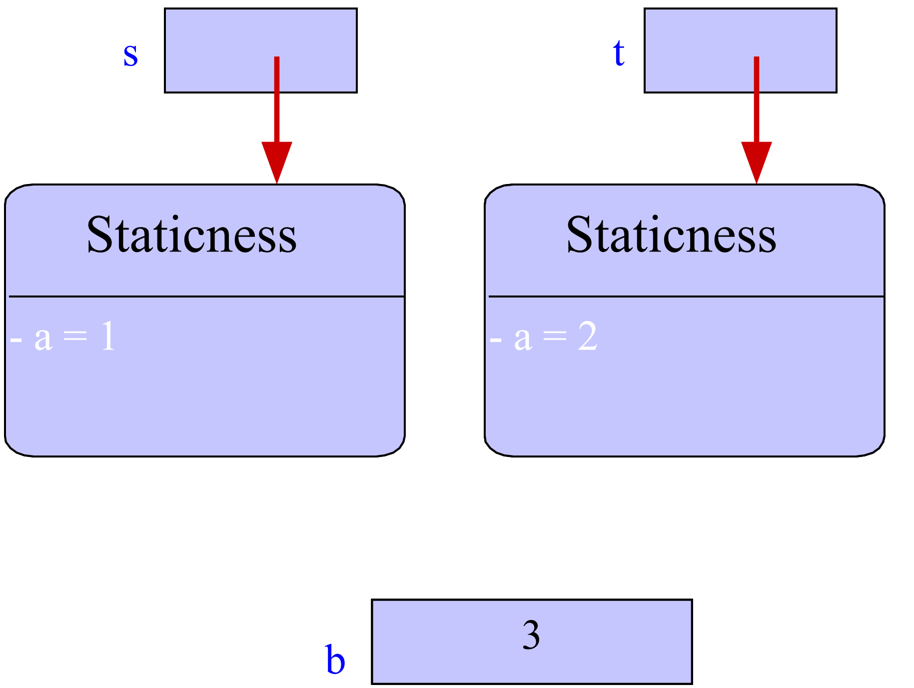
- Output is:

  (a=1,b=3)

  (a=2,b=3)

Static vs. non-static: memory diagram (Read yourself)

```
Staticness s = new Staticness();
Staticness t = new Staticness();
s.increment();
t.increment();
t.increment();
System.out.println (s);
System.out.println (t);
```



Parameter passing

Java parameter passing

- Consider the following code:

```
static void foobar (int y) {
    y = 7;
}

public static void main (String[] args) {
    int x = 5;
    foobar (x);
    System.out.println(x);
}
```

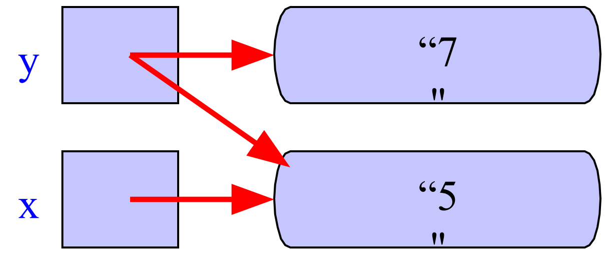
formal parameter

actual parameter

y	5
x	5

- What gets printed?

Java parameter passing



- Consider the following code:

```
static void foobar (String y) {
    y = “7”;
}

public static void main (String[] args) {
    String x = “5”;
    foobar (x);
    System.out.println(x);
}
```

formal parameter

actual parameter

- What gets printed?

Java parameter passing

- Consider the following code:

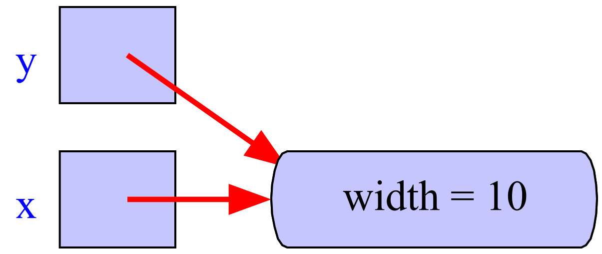
```
static void foobar (Rectangle y) {
    y.setWidth (10);
}
```

formal parameter

```
public static void main (String[] args) {
    Rectangle x = new Rectangle();
    foobar (x);
    System.out.println(x.getWidth());
}
```

actual parameter

- What gets printed?



Java parameter passing

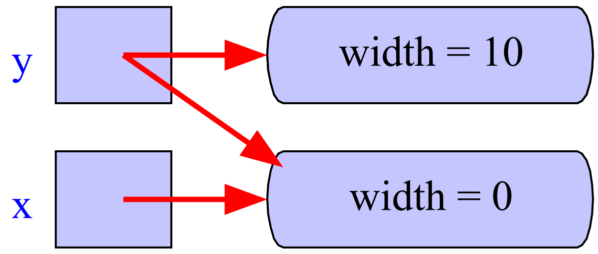
- Consider the following code:

```
static void foobar (Rectangle y) {
        y = new Rectangle();
        y.setWidth (10);
}

public static void main (String[] args) {
        Rectangle x = new Rectangle();
        foobar (x);
        System.out.println(x.getWidth());
}
```

formal parameter

actual parameter

- What gets printed?



Java parameter passing

- Consider the following code:

```
static void swap (int a, int b) {
    int temp=a;a=b;b=temp;
}

public static void main (String[] args) {
    int x=10, y=20;
    swap (x,y);
    System.out.println(x,y);
}
```

formal parameter

actual parameter

- What gets printed?

Java parameter passing

- Consider the following code:

```
static void swap (Integer a, Integer b) {
    Integer temp=a;a=b;b=temp;
}

public static void main (String[] args) {
    Integer x=10, y=20;
    swap(x,y);
    System.out.println(x,y);
}
```

formal parameter

actual parameter

- What gets printed?

x **20**   y **10**

Java parameter passing

- The value of the actual parameter gets copied to the formal parameter
  - This is called pass-by-value
  - C/C++ is also pass-by-value
- Any changes to the formal parameter are forgotten when the method returns

- However, if the parameter is a reference to an object, that object can be modified
  - Similar to how the object a final reference points to can be modified

Value parameter passing demonstration

```
public class ParameterDemo {
    public static double add(double x, double y) {
        double result = x + y;
        return result;
    }
    public static double multiply(double x, double y) {
        x = x * y;
        return x;
    }
    public static void main(String[] args) {
        double a = 8, b = 11;
        double sum = add(a, b);
        System.out.println(a + " + " + b + " = " + sum);
        double product = multiply(a, b);
        System.out.println(a + " * " + b + " = " + product);
    }
}
```

PassingReferences.java

PassingReferences.java

```
Import java.awt.*;
public class PassingReferences {
	public static void f(Point v) {
		v = new Point(0, 0);
	}

	public static void g(Point v) {
		v.setLocation(0, 0);
	}

	public static void main(String[] args) {
		Point p = new Point(10, 10);
		System.out.println(p);

		f(p);
		System.out.println(p);

		g(p);
		System.out.println(p);
	}
}
```

PassingReferences.java run

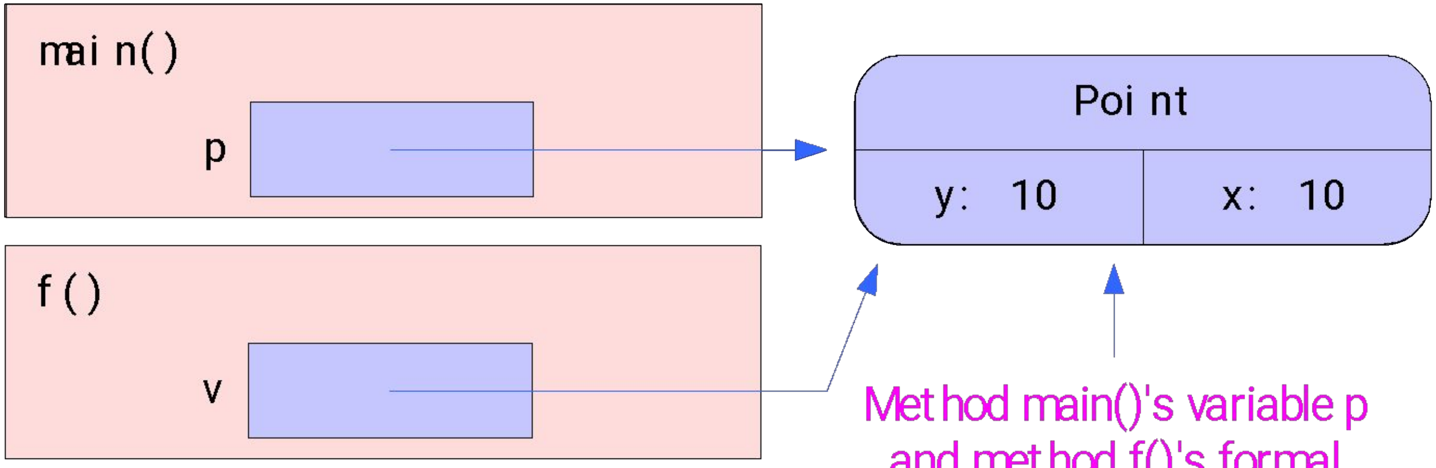
Java Program Design

```
cmd:   javac PassingReferences.java

cmd:   java PassingReferences

java.awt.Point[x=10,y=10]
java.awt.Point[x=10,y=10]
java.awt.Point[x=0,y=0]

cmd:
```

g() can change the attributes of the object to which p refers

PassingReferences.java

```
public static void main(String[] args) {
        Point p = new Point(10, 10);
        System.out.println(p);

        f(p);
```

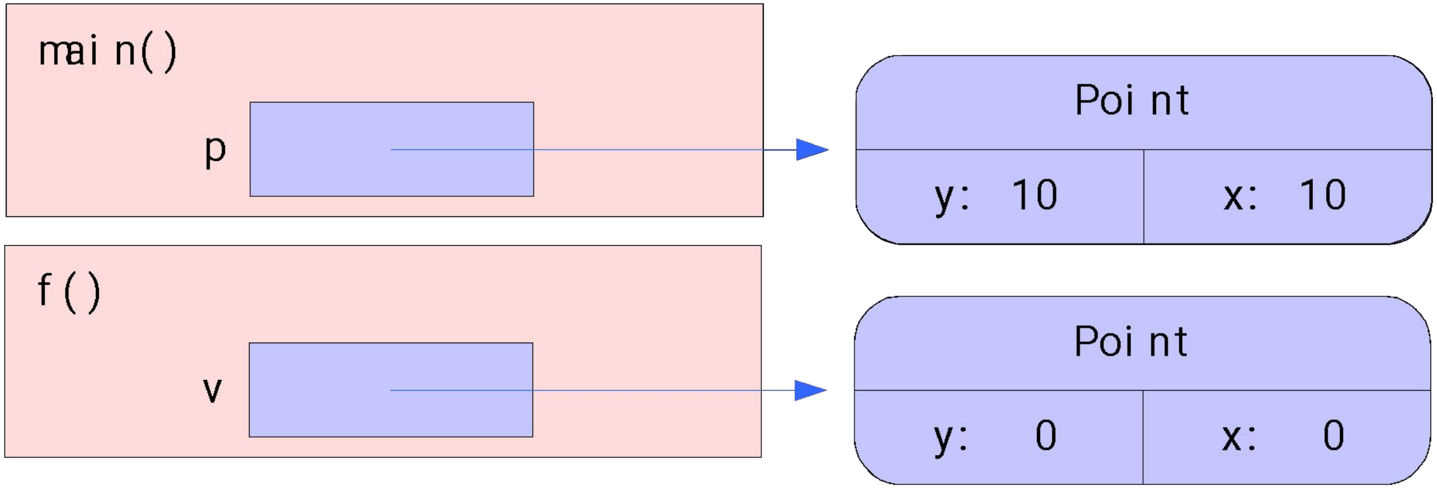


Method main()'s variable p and method f()'s formal parameter v have the same value, which is a reference to an object representing location (10, 10)

java.awt.Point[x=10,y=10]

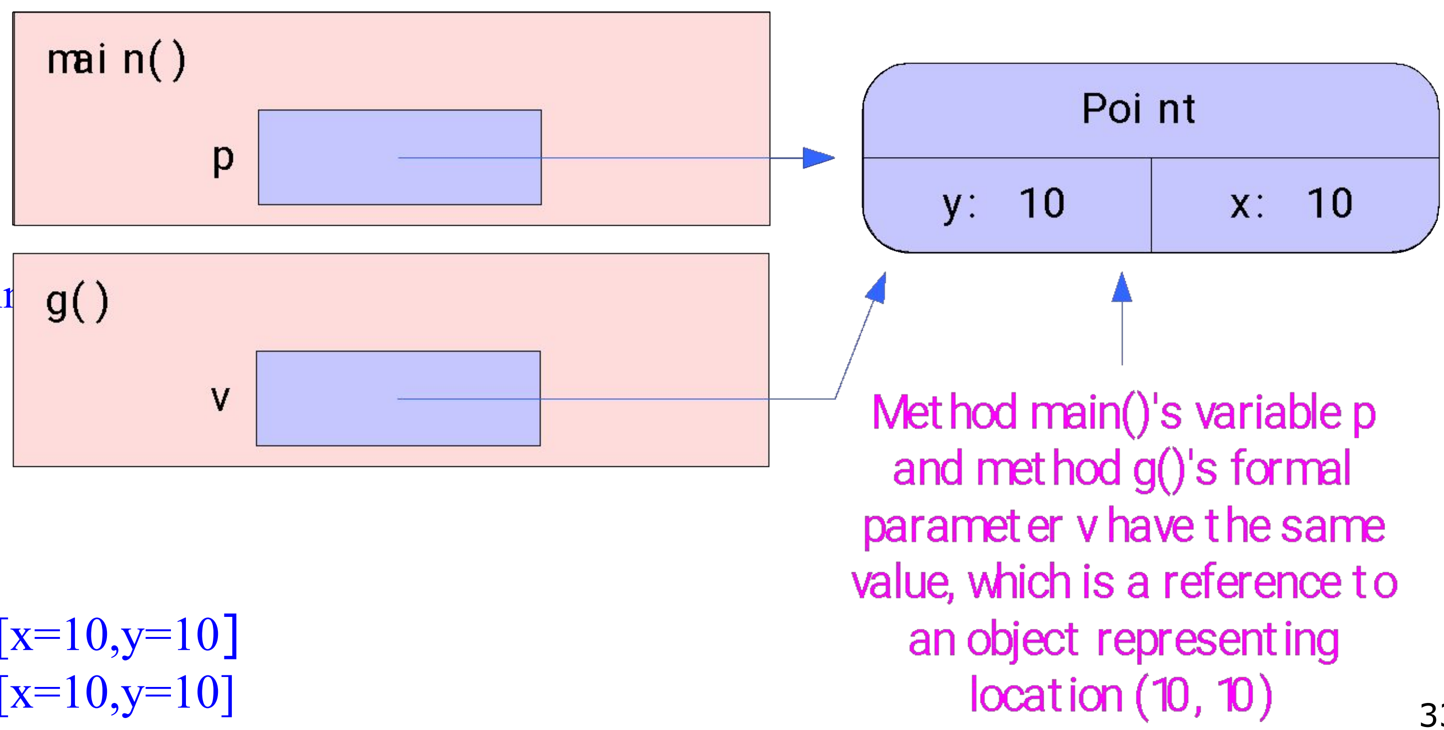
PassingReferences.java

```
public static void f(Point v) {
        v = new Point(0, 0);
}
```



PassingReferences.java

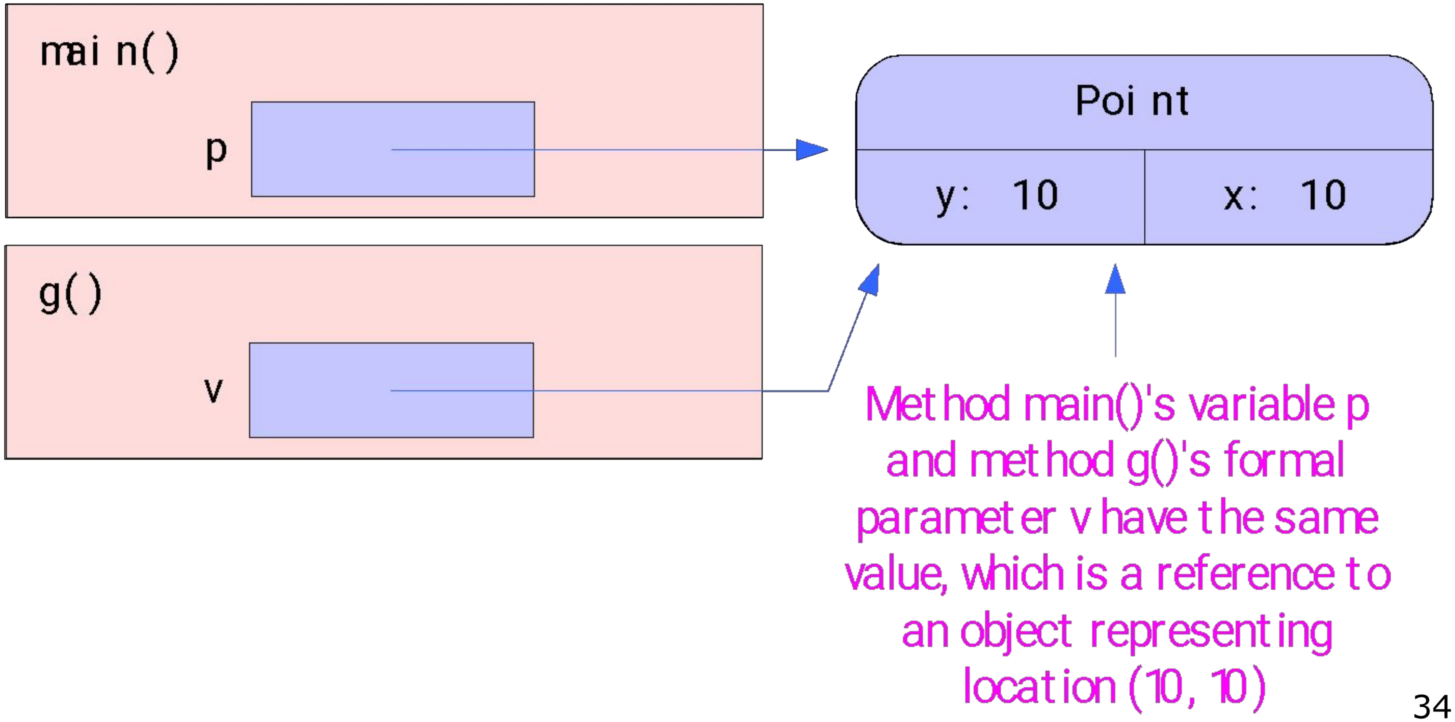
```
public static void main(String[] args) {
        Point p = new Point(10, 10);
        System.out.println(p);

        f(p);

        System.out.prin

        g(p);
```



java.awt.Point[x=10,y=10]
java.awt.Point[x=10,y=10]

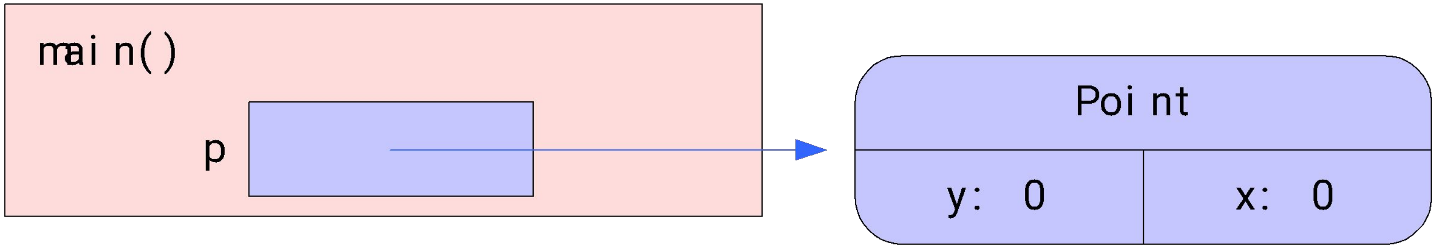
PassingReferences.java

```
public static void g(Point v) {
        v.setLocation(0, 0);
}
```



PassingReferences.java

```
public static void main(String[] args) {
    Point p = new Point(10, 10);
    System.out.println(p);

    f(p);
```




```
    System.out.println(p);

    g(p);
    System.out.println(p);
```

java.awt.Point[x=10,y=10]
java.awt.Point[x=10,y=10]
java.awt.Point[x=0,y=0]

Overloading

Overloading

- Have seen it often before with operators

```
int i = 11 + 28;
double x = 6.9 + 11.29;
String s = "April " + "June";
```

- Java also supports method overloading
  - Several methods can have the same name
  - Useful when we need to write methods that perform similar tasks but different parameter lists
  - Method name can be overloaded as long as its signature is different from the other methods of its class
    - Difference in the names, types, number, or order of the parameters

Legal

```
public static int power(int x, int n) {
	int result = 1;
	for (int i = 1; i <= n; ++i) {
		result *= x;
	}
	return result;
}

public static double power(double x, int n) {
	double result = 1;
	for (int i = 1; i <= n; ++i) {
		result *= x;
	}
	return result;
}
```

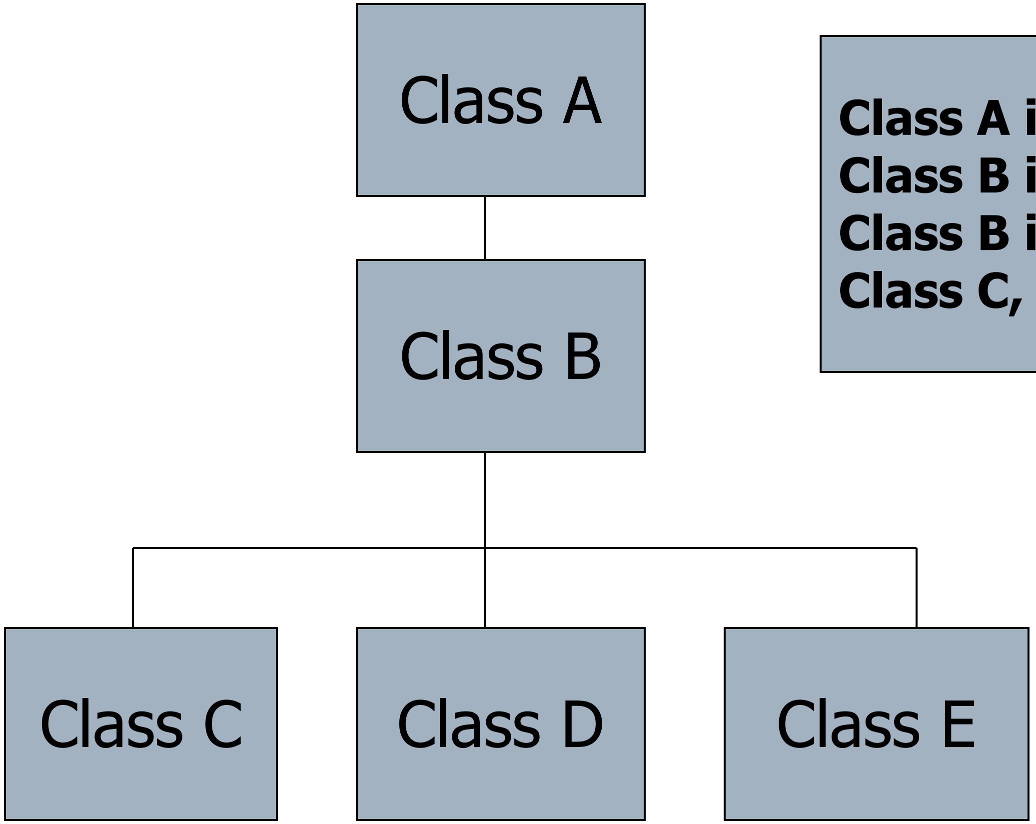
What’s the output?

```
public static void f(int a, int b) {
	System.out.println(a + b);
}

public static void f(double a, double b) {
	System.out.println(a - b);
}

public static void main(String[] args) {
	int i = 19;
	double x = 54.0;

	f(i, x);
}
```

Inheritance

- Inheritance allows us to derive a new class from an existing one
- The existing class is called the *superclass* or *base-class*.
- derived class is called the *subclass* or *derived-class*.
- Instances of the derived class *inherit* all the properties and functionality that is defined in the base class.
- Usually, the derived class adds more functionality and properties.

Inheritance



Class A is the superclass of B
Class B is the subclass of A
Class B is the superclass of C, D, E
Class C, D and E are subclasses of B

Inheritance

- we can build class hierarchies using the keyword extends
- each child (subclass) inherits all the data and methods of its parent (superclass)
- we can add new methods in the subclass, or override the inherited methods
- private data and methods are inherited, but cannot be accessed directly; protected data and methods can be accessed directly
- constructor methods must be invoked in the first line in a subclass constructor as a call to super
- inheritance allows us to re-use classes by specialising them

Inheritance Example 1

```
class BaseClass
{
   public void doSomething()
   {
      System.out.println("BaseClass doSomething");
   }
}
class SubClass extends BaseClass
{
}
public class InheritanceExample1
{
   public static void main(String args[])
   {
      SubClass sc = new SubClass();
      sc.doSomething();
      BaseClass bc = new SubClass();
      bc.doSomething();
   }
}
```

Inheritance Example 2

```
class BaseClass{
    public void doSomething()     {
                System.out.println("BaseClass doSomething");
    }
}
class SubClass extends BaseClass{
    public void doSomething()     {
        System.out.println("SubClass doSomething");
    }
}
public class InheritanceExample2     {
    public static void main(String args[])    {
        SubClass sc = new SubClass();
        sc.doSomething();
        BaseClass bc = new SubClass();
        bc.doSomething();
    }
}
```

Example 3 (with super)

```
class BaseClass{
    public void doSomething(){
            System.out.println("BaseClass doSomething");
    }
}
class SubClass extends BaseClass{
    public void doSomething()    {
        System.out.print("Super: ");
        super.doSomething();
        System.out.println("SubClass doSomething");
    }
}
```

Inheritance: a Basis for Code Reusability

- **Fast implementation** - we need not write the implementation of `all classes` from scratch, we just implement the additional functionality.
- **Ease of use** - If someone is already familiar with the base class, then the derived class will be easy to understand.
- **Less debugging** - debugging is restricted to the additional functionality.
- **Ease of maintenance** - if we need to correct/improve the implementation of `base class`, `derive class` is automatically corrected as well.
- **Compactness** - our code is more compact and is easier to understand.

Rules of Overriding

- When you derive a class B from a class A, the interface of class B will be a *superset* of that of class A (except for constructors)
- You cannot remove a method from the interface by sub-classing
- However, class B can *override* some of the methods that it inherits and thus change their functionality.
- The over-ridden methods of the super-class are no longer accessible from a variable of the sub-class type.
- They can be invoked from within the sub-class definition using the super.method(…) syntax.
- The *contract* of a method states what is expected from an overriding implementation of the method.