Programming using Java

Java Classes and Objects: A Preview

Objectives:

- Get an introduction to Classes and Objects
- Get a general idea of how a small program is put together

Classes and Source Files

- A class defines a class of objects.

Convention:

a class name starts with a <u>capital letter</u>

Class name:	File name:
SomeClass	SomeClass.java
Ramblecs	Ramblecs.java
FallingCube	FallingCube.java

Same upper / lower case letters

Files and Folders

- javac automatically looks for classes (.java or .class files) in the current folder, or, if classpath is set, in folders listed in the classpath string.
- A missing file may be reported as a syntax error when compiling another file.
- If you set classpath, include the current folder. It is denoted by a dot. For example:

  .;C:\javamethods\EasyIO

- IDE helps take care of the file locations.

public class *SomeClass*

{

- *Fields* — Attributes / variables that define the object's state. Can hold numbers, characters, strings, other objects. Usually private.
- *Constructors* — Code for constructing a new object and initializing its fields. Usually public.
- *Methods* — Actions that an object can take. Can be public or private.

}

private: visible only inside this class
public: visible in other classes

```
public class FallingCube
{
  private final int cubeSize;
  private int cubeX, cubeY;   // Cube coordinates
  ...
  private char randomLetter;  // Cube letter

  public FallingCube(int size)
  {
    cubeSize = size;
    ...
  }

  public void start()
  {
    cubeX = 0;
    cubeY = -cubeSize;
    ...
  }
  ...
}
```

Fields

Constructor

The name of a constructor is always the same as the name of the class.

Methods

Fields

Fields

You name it!

private (or public) [static] [final]

datatype name;

Usually private

May be present: means the field is shared by all objects in the class

May be present: means the field is a constant

int, double, etc., or an object: String, JButton, FallingCube, Timer

Field qualifier

- Form of field declaration
  - **[qualifier] DataType fieldNames;**
    - qualifier : public, protected, private, static, final, volatile

```
int  anInteger, anotherIntegrer;
public String usage;
static long idNum = 0;
public static final double earthWeight = 5.97e24;
```

Field

- Access Modifier
  - Access Permission Level from other classes
  - public, protected, private
  - *Default/no declaration*: package

Modifier	Class	Sub-Class	Same Package	All Class
private	O	X	X	X
package	O	X	O	X
protected	O	O	O	X
public	O	O	O	O

Fields

- May have *primitive data types*:
  int, char, double, etc.

```
private int cubeX, cubeY;      // cube coordinates
...
private char randomLetter;    // cube letter
```

Fields

- May be <u>objects</u> of different types:

```
private FallingCube cube;
private Timer t;
private static final String letters;
```

Constructors

Constructors

- Constructors are like methods for creating objects of a class.
- Most constructors initialize the object's fields.
- Constructors may take parameters.
- A class may have several constructors that differ in the number or types of their parameters.
- All of a class's constructors have the same name as the class.

Java™ 2 Platform
Std. Ed. v1.3

All Classes

JarInputStream
JarOutputStream
JarURLConnection
JButton
JCheckBox
JCheckBoxMenuItem

Constructors

Constructor Summary

JButton() Creates a button with no set text or icon.
JButton(Action a) Creates a button where properties are taken from the Action supplied.
JButton(Icon icon) Creates a button with an icon.
JButton(String text) Creates a button with text.
JButton(String text, Icon icon) Creates a button with initial text and an icon.

```
go = new JButton("Go");
```

Constructors (cont'd)

- Call them using the new operator:

```
cube = new FallingCube(CUBESIZE);
```

Calls FallingCube's constructor with CUBESIZE as the parameter

```
...
```

```
t = new Timer(delay, this)
```

Calls Timer's constructor with delay and this (i.e. this object) as the parameters (see Java docs for javax.swing.Timer)

Class constructors

- The purpose of the constructor is to initialize the instance variables
  - Default constructor
  - Constructor overloading

** forgetting to call a constructor

** constructor is invoked only when an object is first created. You cannot call the constructor to reset an object.

Constructors

- If a class has more than one constructor, they are “overloaded” and must have different numbers and/or types of arguments.
- Programmers often provide a “no-args” constructor that takes no arguments.
- If a programmer does not define <u>any</u> constructors, Java provides one default no-args constructor, which allocates memory and sets fields to the default values.

Constructors (cont'd)

```
public class Fraction
{
   private int num, denom;

   public Fraction ( )
   {
      num = 0;
      denom = 1;
   }

   public Fraction (int n)
   {
      num = n;
      denom = 1;
   }

   public Fraction (int p, int q)
   {
      num = p;
      denom = q;
      reduce ();
   }

   public Fraction (Fraction other)
   {
      num = other.num;
      denom = other.denom;
   }
   ...
}
```

"No-args" constructor

Copy constructor

Constructors

- Constructors of a class can call each other using the keyword this — a good way to avoid duplicating code:

```
public class Fraction
{
   ...

   public Fraction (int n)
   {
      this (n, 1);
   }
   ...
```

```
   ...
   public Fraction (int p,
int q)
   {
      num = p;
      denom = q;
      reduce ();
   }
   ...
```

Operator new

- Constructors are invoked using new

```
Fraction  f1 = new Fraction ( );
Fraction  f2 = new Fraction (5);
Fraction  f3 = new Fraction (4, 6);
Fraction  f4 = new Fraction (f3);
```

0 / 1

5 / 1

2 / 3

2 / 3

Operator new (cont'd)

- You must create an object before you can use it; the new operator is a way to do it

```
Fraction  f;
```

f is set to null

```
f = new Fraction (2, 3);
```

Now f refers to a valid object

```
f = new Fraction (3, 4);
```

Now f refers to another object (the old object is "garbage-collected")

The this Keyword

this Keyword

this.varname

Invocation of Constructors

this(args);

```
class Point3D {
  double x;
  double y;
  double z;
  Point3D(double x, double y, double z)
{
    this.x = x;
    this.y = y;
    this.z = z;
  }
}
```

```
class ThisKeywordDemo {
  public static void main(String args[])
{
    Point3D p = new Point3D(1.1, 3.4, -2.8);
    System.out.println("p.x = " + p.x);
    System.out.println("p.y = " + p.y);
    System.out.println("p.z = " + p.z);
  }
}
```

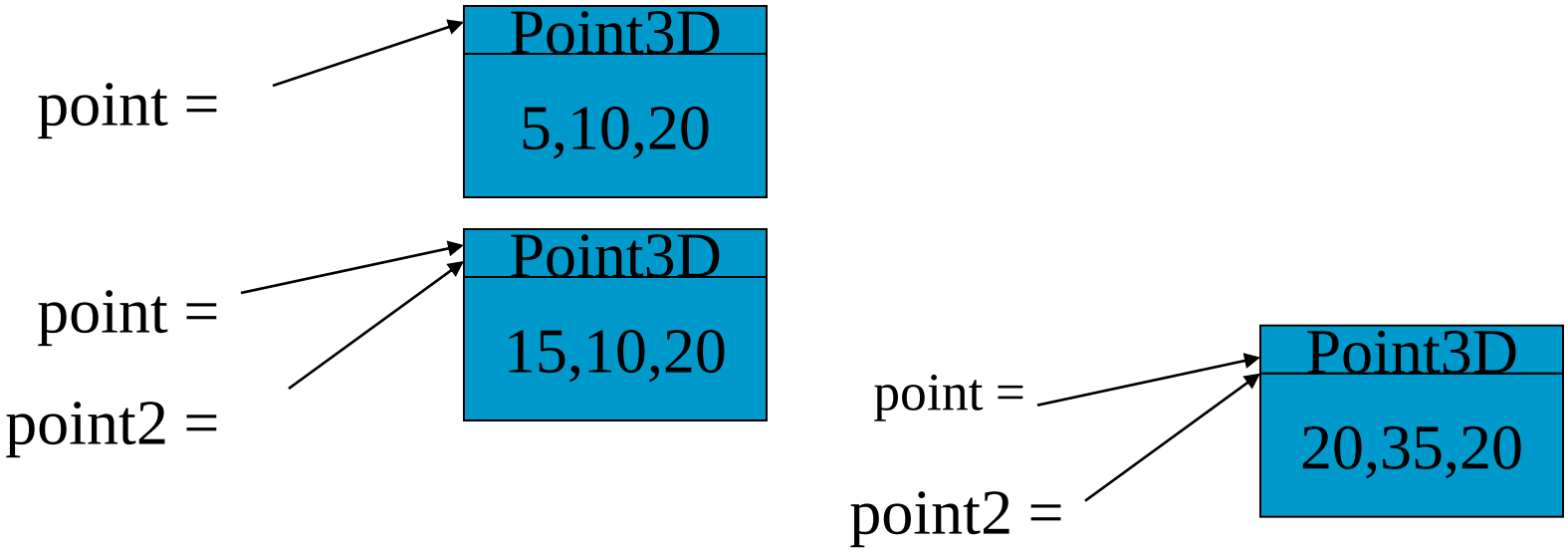
Calling one constructor from another

```
public class OOPClass
{
    int no_of_stud;
        public OOPClass( int i)
        {
            no_of_stud = i;
        }
        public OOPClass()
        {
            this(50);
        }
}
```

this(…) has to be the first line in the constructor

Object references

- A variable holds a memory location of an object.



null references

- The null value is unique in that it can be assigned to a variable of any reference type whatsoever.
- Ex:
  - String middleInitial = null;
  - Car c = null;
- Compare objects
  - if (c == null)……

Methods

Methods

- Call them for a particular object:

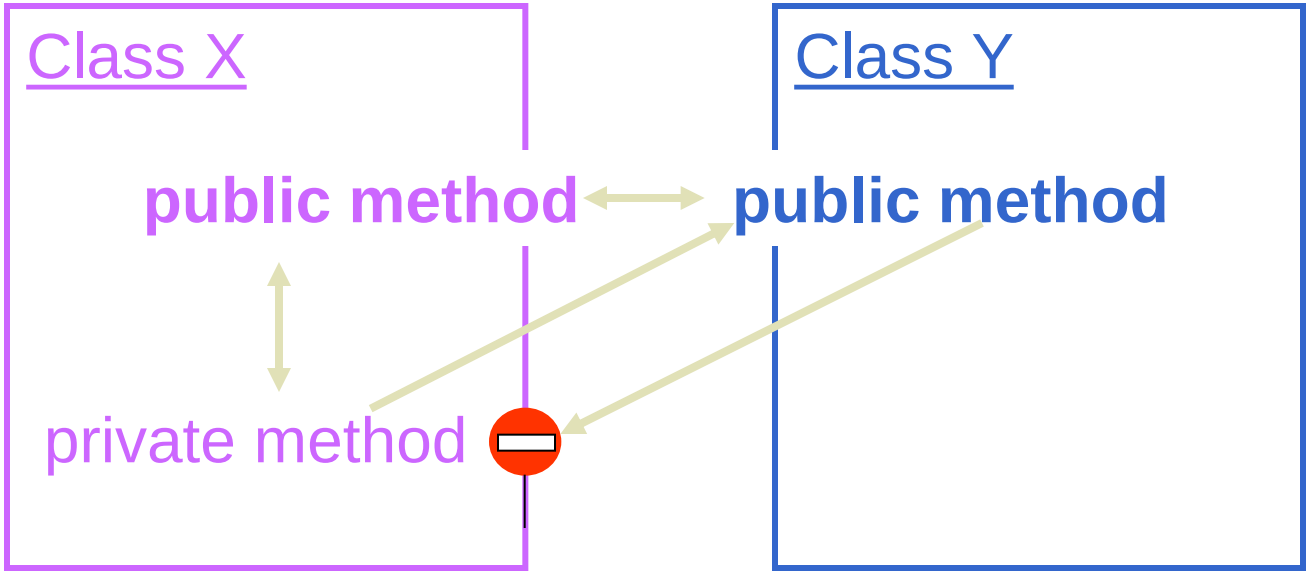
  cube.start();

But call *static* (“*class*”) *methods* for the whole class, not a specific object:

y = **Math.**sqrt (x);

Methods

- Constructors and methods can call other public and private methods of the same class.
- Constructors and methods can call only public methods of another class.



Methods (cont'd)

- You can call methods with specific arguments:

```
g.drawRect (75, 25, 150, 50);
g.drawString ("Welcome", 120, 50);
```

- The number and types of arguments must match the method's parameters:

```
public void drawRect ( int x, int y,
                                int width, int height )   {...}
public void drawString ( String msg, int x, int y  )   {...}
```

Methods

```
public [or private]  returnType
                     methodName (type1 name1, ..., typeN nameN)
{
   ...
}
```

Body

Header

- To define a method:
  - decide between public and private (usually public)
  - give it a name
  - specify the types of arguments (formal parameters) and give them names
  - specify the method's return type or chose void
  - write the method's code

Methods (cont'd)

- A method is always defined inside a class.
- A method returns a value of the specified type unless it is declared void; the return type can be any primitive data type or a class type.
- A method's arguments can be of any primitive data types or class types.

Empty parentheses indicate that a method takes no arguments.

```
public [or private]  returnType  methodName ( )
{    ...   }
```

Methods: Java Style

- Method names start with lowercase letters.
- Method names usually sound like verbs.
- The name of a method that returns the value of a field often starts with get:

  getWidth, getX

  The name of a method that sets the value of a field often starts with set:

  setLocation, setText

Method

- Form of Method Declaration

[qualifier] returnType
methodName(parameterList) {
// method body
}

  - **qualifier : modifier**, static, final, native, synchronized
  - **returnType : void unless return value**

```
class MethodExample {
        int simpleMethod() {
                //...
        }
        public void emptyMethod()  {  }
}
```

Method

- Method Qualifier
  - Access Modifier
    - Access Permission Level to Method from Other Class
    - Same as that of access modifier in field
  - **static**
    - static method, class method
    - Same role of Global function
    - Use only the static field of correspond class or the static method
    - Can be referred by only class name

```
ClassName.methodName;
```

Method

- **final**
  - Final method
  - Method which cannot be redefined in subclass

- **synchronized**
  - Synchronization method
  - Control the thread so that only on thread can always access the target
- **native**
  - To use the implementation written in other programming languages such as C language

Parameter

- Parameter Passing
  - Formal parameter
  - Actual parameter

```
void parameterPass(int i, Fraction f) {
    // ...
}
```

- Local variable referred in method

```
class Fraction {
    int numerator, denominator;                          // Field
    public Fraction(int numerator, int denominator) {  // Parameter
        // ...
    }
}
```

Parameter

- Call by value
- Call by reference
- main method

```
public static void main(String[] args) {
    // …
}
```

main()

- Pass in command line
  - public static void main(**<u>String[ ] args</u>**)

[command line]

		args[0]	**args[1]**	**args[2]**
java	ClassName	<u>args1</u>	<u>args2</u>	<u>args3</u>

Overloaded Methods

- Methods of the <u>same</u> class that have the same name but different numbers or types of arguments are called ***overloaded methods.***
- Use overloaded methods when they perform similar tasks:

```
public void move (int x, int y)   { ... }
public void move (double x, double y)
{ ... }
public void move (Point p)   { ... }

public Fraction add (int n)   { ... }
public Fraction add (Fraction other)   {
... }
```

Overloaded Methods (cont'd)

- The compiler treats overloaded methods as completely different methods.
- The compiler knows which one to call based on the number and the types of the arguments:

```
public class Circle
{
   ...
    public void move (int x, int y)
     { ... }

    public void move (Point p)
     { ... }
   ...
}
```

```
Circle circle =  new Circle(5);

circle.move (50, 100);
...
Point center =
            new Point(50, 100);
circle.move (center);
...
```

Method Overloading

- Case of the same method name, but different in no. of parameter and type

```
void  methodOver(int i)  {  /* . . . */  }          // the first form
 void  methodOver(int i, int j){  /* . . . */ }// the second form
```

  - In case of method overloading, compilers do the following :
    - ✶Seek the method having the same parameter type
    - Seek the method having the parameter which can be converted by basic type casting

Method Overloading

```
public class  MethodOver {
        void someThing() {        // ...
       }
       void someThing(int i) {    // …
       }
       void someThing(int i, int j) {    // …
       }
       public static void main(String[] args) {
             MethodOver  m = new MethodOver();
             m.someThing();
             m.someThing(526);
             m.someThing(54, 526);
       }
}
```