Programming using Java

Java Classes and Objects:
A Preview

Methods

Methods

- Call them for a particular object:

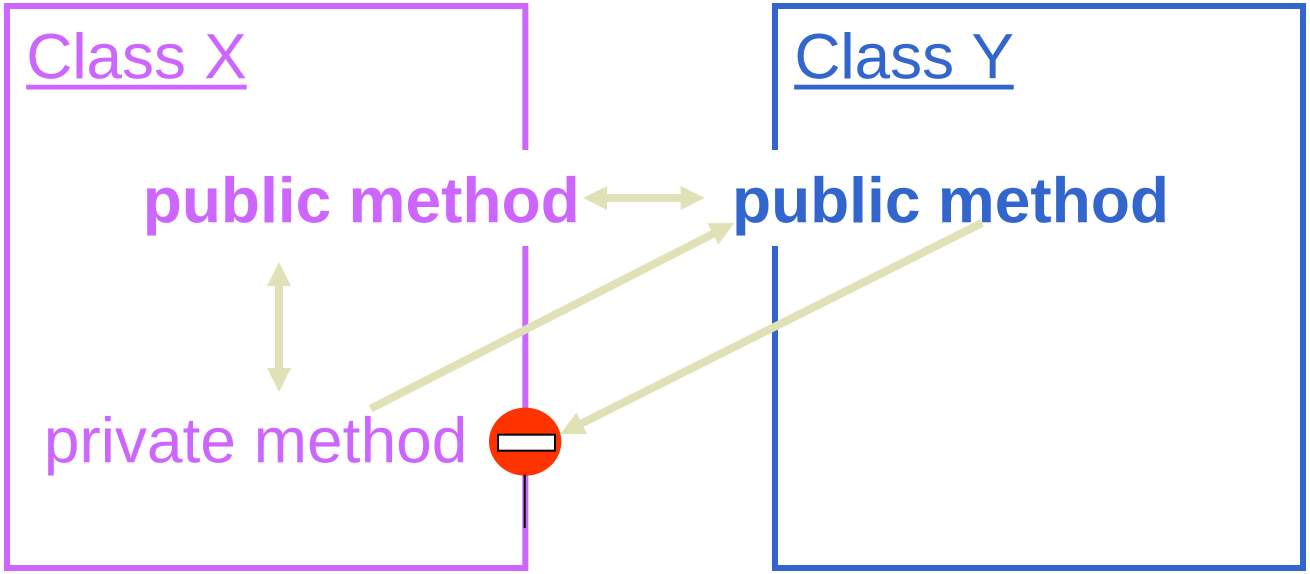
```
cube.start();
```

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

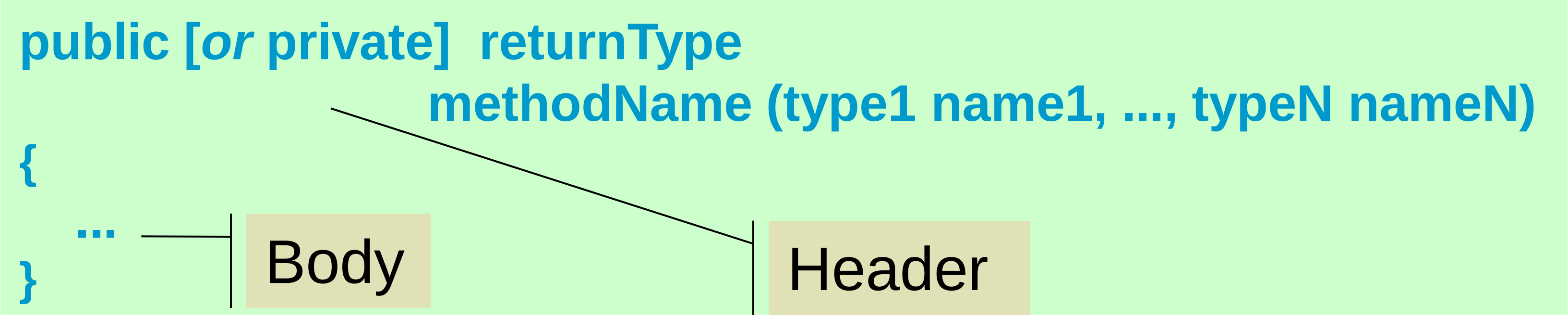
```
y = Math.sqrt (x);
```

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



```
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 threads so that only one 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(**String[ ] args**)

```
[command line]              args[0]  args[1]   args[2]
     java  ClassName   args1  args2  args3
```

Overloaded Methods

- Methods of the same 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);
        }
}
```

Static

Static Fields

- A *static* field (a.k.a. *class field* or *class variable*) is shared by all objects of the class.

- A static field can hold a constant shared by all objects of the class:

```
public class RollingDie
{
   private static final double slowDown = 0.97;
   private static final double speedFactor = 0.04;
   ...
```

Reserved words:
static
final

- A non-static field (a.k.a. *instance field* or *instance variable*) belongs to an individual object.

Static Fields (cont’d)

- Static fields are stored with the class code, separately from non-static fields that describe an individual object.

- Public static fields, usually global constants, are referred to in other classes using “dot notation”:
  `ClassName.constName`

```
double area = Math.PI * r * r;
setBackground(Color.blue);
c.add(btn, BorderLayout.NORTH);
System.out.println(area);
```

Static Fields (cont'd)

- Usually static fields are NOT initialized in constructors (they are initialized either in declarations or in public static methods).
- If a class has only static fields and does not have any non-static (instance) fields, there is no point in creating objects of that class (all of them would be identical).
- Math and System are examples of the above. In fact, they have no public constructors and cannot be *instantiated.*

Static Methods

- Static methods can access and manipulate a class's static fields.
- Static methods cannot access non-static fields or call non-static methods of the class.
- Static methods are called using "dot notation": ClassName.statMethod(...)

```
double x = Math.random();
double y = Math.sqrt (x);
System.exit();
```

Instance Methods

- Non-static methods are also called *instance methods.*
- An instance method is called for a particular object using “dot notation”:

```
objName.instMethod(...);
```

- Instance methods can access ALL fields and call ALL methods of their class — both class and instance fields and methods.

Static (Class) vs. Non-Static (Instance)

```
public class MyClass
{
   public static final int statConst;
   private static int statVar;

   private int instVar;
   ...
   ...

   public static int statMethod(...)
   {
      statVar = statConst;
      statMethod2(...);
```

OK

```
      instVar = ...;
      instMethod(...);
```

Error!

```
   }
```

```
   public int instMethod(...)
   {
```

All OK

```
      statVar = statConst;
      inst Var = statConst;
      instVar = statMethod(...);
      statVar = instMethod2(...);
     ...
   }

   public int instMethod2(...)
   {
      ...
   }
   ...
}
```

Static vs. Non-Static (cont’d)

- Note: main is static and therefore cannot access non-static fields or call non-static methods of its class:

```
public class Hello
{
   private String message = "Hello,
World";

   public static void main (String[ ]
args)
   {
      System.out.println (message);
   }
}
```

Error: non-static variable message is used in static context (main)

Static Initialization Statement

- The Statement to be executed at the same time when the system initialize the static variable in the class

- From

static { <statement> **}**

Static Initialization Statement

- Execution Order
  - Order of initialization of static init. Statement and static variable : existing order in the program

```
class Initializers {
        static {   i = j + 2; }        // Error
        static int i, j;
        static j = 4;
        //...
 }
```

  - Executed earlier than constructor

finalize Method

- Garbage Collector
  - Automatic Memory Management

- finalize Method
  - Call the finalize method before the garbage collector reclaim the memory
- Provide the method to release the resources
  - Programmer can remove the resources(ex:open files) directly using finalize method which garbage collector cannot reclaim.

return

- A method, unless void, returns a value of the specified type to the calling method.
- The return statement is used to immediately quit the method and return a value:

```
return expression;
```

The type of the return value or expression must match the method's declared return type.

return

- A method can have several return statements; then all but one of them must be inside an if or else (or in a switch):

```
public someType myMethod (...)
{
    ...
    if (...)
        return <expression1>;
    else
        return <expression2>;
    ...
    return <expression3>;
}
```

return

- A boolean method can return true, false, or the result of a boolean expression:

```
public boolean myMethod (...)
{
   ...
   if (...)
      return true;
   ...
   return n % 2 == 0;
}
```

return

- A void method can use a return statement to quit the method early:

```
public void myMethod (...)
{
    ...
    if (...)
        return;
    ...
}
```

No need for a redundant return at the end

return

- If its return type is a class, the method returns a reference to an object (or null).
- Often the returned object is created in the method using new. For example:

```
public Fraction inverse ()
{
    if (num == 0)
        return null;
    return new Fraction (denom, num);
}
```

- The returned object can also come from the arguments or from calls to other methods.

Encapsulation

- Hiding the implementation details of a class (making all fields and helper methods private) is called ***encapsulation***.
- Encapsulation helps in program maintenance and team development.
- A class encapsulates a small set of well-defined tasks that objects of a class can perform.

The main Method

The Main Method - Concept

- **main** method
  - the system locates and runs the main method for a class when you run a program
  - other methods get execution when called by the main method explicitly or implicitly
  - must be public, static and void

The Main Method - Getting Input from the Command Line

- When running a program through the `java` command, you can provide a list of strings as the real arguments for the `main` method. In the `main` method, you can use `args[index]` to fetch the corresponding argument

```
class Greetings {
  public static void main (String args[]){
    String name1 = args[0];
    String name2 = args[1];
    System.out.println("Hello " + name1 + “&“ +name2);
  }
}
```

```
➢ java Greetings Jacky Mary
Hello Jacky & Mary
```

- Note: What you get are strings! You have to convert them into other types when needed.