



## CHAPTER 5

# Defining Classes

Shin-Jie Lee (李信杰)

Assistant Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



# Introduction

---

- ❑ Classes are the most important language feature that make *object-oriented programming (OOP)* possible
- ❑ Programming in Java consists of defining a number of classes
  - Every program is a class
  - All helping software consists of classes
  - All programmer-defined types are classes
- ❑ Classes are central to Java



# Class Definitions

---

- ❑ You already know how to use classes and the objects created from them, and how to invoke their methods
  - For example, you have already been using the predefined **String** and **Scanner** classes
- ❑ Now you will learn how to define your own classes and their methods, and how to create your own objects from them



# A Class Is a Type

---

- ❑ A class is a special kind of programmer-defined type, and variables can be declared of a class type
- ❑ A value of a class type is called an object or *an instance of the class*
  - If Cat is a class, then the phrases “cat is of type Cat,” “cat is an object of the class Cat,” and “cat is an instance of the class Cat” mean the same thing
- ❑ A class determines the types of data that an object can contain, as well as the actions it can perform



# Primitive Type Values vs. Class Type Values

---

- ❑ A primitive type value is a **single piece of data**
- ❑ A class type value or object can have **multiple pieces of data**, as well as actions called *methods*
  - All objects of a class have the same methods
  - All objects of a class have the same pieces of data (i.e., name, type, and number)
  - For a given object, each piece of data can hold a different value



# The Contents of a Class Definition

---

- ❑ A class definition specifies the data items and methods that all of its objects will have
- ❑ These data items and methods are sometimes called *members* of the object
- ❑ **Data items are called *fields* or *instance variables***
- ❑ Instance variable declarations and method definitions can be placed in any order within the class definition



# Lab

---

```
public class Duck {  
  
    public boolean canfly = false;  
  
    public void quack(){  
        System.out.println("Quack!!");  
    }  
  
}
```



# The new Operator

---

- ❑ An object of a class is named or declared by a variable of the class type:

```
ClassName classVar;
```

- ❑ The **new** operator must then be used to create the object and associate it with its variable name:

```
classVar = new ClassName();
```

- ❑ These can be combined as follows:

```
ClassName classVar = new ClassName();
```





# Lab

---

```
public class Farm {  
  
    public static void main(String[] args) {  
        Duck duck = new Duck();  
    }  
  
}
```



# Instance Variables and Methods

---

- ❑ Instance variables can be defined as in the following two examples
  - Note the **public** modifier (for now):  
**public String instanceVar1;**  
**public int instanceVar2;**
- ❑ In order to refer to a particular instance variable, preface it with its object name as follows:  
**objectName.instanceVar1**  
**objectName.instanceVar2**



# Instance Variables and Methods

- ❑ Method definitions are divided into two parts: a *heading* and a *method body*:

```
public void myMethod() ← Heading
{
    code to perform some action
    and/or compute a value    } Body
}
```

- ❑ Methods are invoked using the name of the calling object and the method name as follows:

```
classVar.myMethod();
```

- ❑ Invoking a method is equivalent to executing the method body



# Lab

---

```
public class Farm {  
  
    public static void main(String[] args) {  
  
        Duck duck = new Duck();  
  
        boolean canTheDuckFly = duck.canfly;  
        if(canTheDuckFly == true){  
            System.out.println("The duck can fly");  
        }  
  
        duck.quack();  
    }  
}
```



# File Names and Locations

---

- ❑ Reminder: a Java file must be given the same name as the class it contains with an added **.java** at the end
  - For example, a class named **MyClass** must be in a file named **MyClass.java**
- ❑ For now, your program and all the classes it uses should be in the same directory or folder



# More About Methods

---

- ❑ There are two kinds of methods:
  - **Methods that compute and return a value**
  - **Methods that perform an action**
    - This type of method does not return a value, and is called a **void** method
- ❑ Each type of method differs slightly in how it is defined as well as how it is (usually) invoked



# More About Methods

---

- ❑ A method that returns a value must specify the type of that value in its heading:

```
public typeReturned methodName(paramList)
```

- ❑ A **void** method uses the keyword **void** in its heading to show that it does not return a value :

```
public void methodName(paramList)
```



# **main is a void Method**

---

- ❑ A program in Java is just a class that has a **main** method
- ❑ When you give a command to run a Java program, the run-time system invokes the method **main**
- ❑ Note that **main** is a **void** method, as indicated by its heading:  

```
public static void main(String[] args)
```





# return Statements

- ❑ The body of both types of methods contains a list of declarations and statements enclosed in a pair of braces

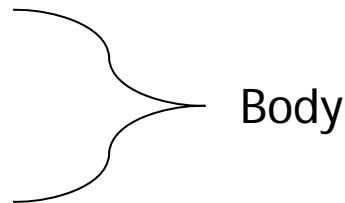
```
public <void or typeReturned> myMethod()
```

```
{
```

declarations

statements

```
}
```





# return Statements

---

- ❑ The body of a method that returns a value must also contain one or more **return** statements
  - A **return** statement specifies the value returned and ends the method invocation:  
**return Expression;**
  - **Expression** can be any expression that evaluates to something of the type returned listed in the method heading



# return Statements

---

- ❑ A **void** method need not contain a **return** statement, unless there is a situation that requires the method to end before all its code is executed
- ❑ In this context, since it does not return a value, a **return** statement is used without an expression:  
**return;**



# Method Definitions

---

- ❑ An invocation of a method that returns a value can be used as an expression anyplace that a value of the **TypeReturned** can be used:

```
TypeReturned tRVariable;  
tRVariable = objectName.methodName();
```

- ❑ An invocation of a **void** method is simply a statement:

```
objectName.methodName();
```



## Any Method Can Be Used As a **void** Method

---

- ❑ A method that returns a value can also perform an action
- ❑ If you want the action performed, but do not need the returned value, you can invoke the method as if it were a **void** method, and the returned value will be discarded:

**objectName.returnValueMethod( );**



# Lab

---

```
public class Duck {  
  
    public boolean canfly = false;  
  
    public void quack(){  
        System.out.println("Quack!!");  
    }  
  
    public String eat(String food){  
        String message = "Thank you! The " + food + " is good!";  
        return message;  
    }  
}
```



# Lab

---

```
public class Farm {  
  
    public static void main(String[] args) {  
        Duck duck = new Duck();  
  
        boolean canTheDuckFly = duck.canfly;  
        if(canTheDuckFly == true){  
            System.out.println("The duck can fly");  
        }  
  
        duck.quack();  
  
        String food = "Hamburger";  
        String message = duck.eat(food);  
        System.out.println(message);  
    }  
}
```



# Local Variables

- ❑ A variable declared within a method definition is called a *local variable*
  - All variables declared in the **main** method are local variables
  - All method parameters are local variables
- ❑ If two methods each have a local variable of the same name, they are still two entirely different variables

```
public class Duck {  
    public boolean canfly = false;  
  
    public String eat(String food){  
        String message = "Thank you! The " + food + " is good!";  
        return message;  
    }  
}
```

instance variable

local variable





# Blocks

---

- ❑ A *block* is another name for a compound statement, that is, a set of Java statements enclosed in braces, `{ }`
- ❑ A variable declared within a block is local to that block, and cannot be used outside the block
- ❑ Once a variable has been declared within a block, its name cannot be used for anything else within the same method definition



## Declaring Variables in a **for** Statement

- ❑ You can declare one or more variables within the initialization portion of a **for** statement
- ❑ A variable so declared will be local to the **for** loop, and cannot be used outside of the loop
- ❑ If you need to use such a variable outside of a loop, then declare it outside the loop

```
int sum = 0;
for(int i=1; i <= 50; i++)
{
    sum += i;
}
System.out.println("The total is: " + sum);
```



# Parameters of a Primitive Type

---

- ❑ The methods seen so far have had no parameters, indicated by an empty set of parentheses in the method heading
- ❑ Some methods need to receive additional data via a list of *parameters* in order to perform their work
  - These *parameters* are also called *formal parameters*



# Parameters of a Primitive Type

---

- ❑ A parameter list provides a description of the data required by a method
  - It indicates the number and types of data pieces needed, the order in which they must be given, and the local name for these pieces as used in the method

```
public double myMethod(int p1, int p2, double p3)
```



# Parameters of a Primitive Type

---

- ❑ When a method is invoked, the appropriate values must be passed to the method in the form of *arguments*
  - Arguments are also called *actual parameters*
- ❑ The number and order of the arguments must exactly match that of the parameter list
- ❑ The type of each argument must be compatible with the type of the corresponding parameter

```
int a=1,b=2,c=3;  
double result = myMethod(a,b,c);
```



# Parameters of a Primitive Type

---

- ❑ In the preceding example, the value of each argument (not the variable name) is plugged into the corresponding method parameter
  - This method of plugging in arguments for formal parameters is known as the *call-by-value mechanism*



# Parameters of a Primitive Type

---

- ❑ If argument and parameter types do not match exactly, Java will attempt to make an automatic type conversion
  - In the preceding example, the `int` value of argument `c` would be cast to a `double`
  - A primitive argument can be automatically type cast from any of the following types, to any of the types that appear to its right:

`byte`→`short`→`int`→`long`→`float`→`double`  
`char` \_\_\_\_\_↑



# Parameters of a Primitive Type

---

- ❑ A parameter is often thought of as a blank or placeholder that is filled in by the value of its corresponding argument
- ❑ However, a parameter is more than that: it is actually a local variable
- ❑ When a method is invoked, the value of its argument is computed, and the corresponding parameter (i.e., local variable) is initialized to this value
- ❑ Even if the value of a formal parameter is changed within a method (i.e., it is used as a local variable) the value of the argument cannot be changed





# Lab

---

```
public class Duck {  
  
    public boolean canfly = false;  
  
    public void quack(){  
        System.out.println("Quack!!");  
    }  
  
    public String eat(String food){  
        String message = "Thank you! The " + food + " is good!";  
        return message;  
    }  
  
    public void swim(int distance){  
        distance = distance - 1;  
        System.out.println("The distance of my swimming is " + distance);  
    }  
}
```



# Lab

---

```
public class Farm {  
  
    public static void main(String[] args) {  
        Duck duck = new Duck();  
  
        boolean canTheDuckFly = duck.canfly;  
        if(canTheDuckFly == true){  
            System.out.println("The duck can fly");  
        }  
  
        duck.quack();  
  
        String food = "Hamburger";  
        String message = duck.eat(food);  
        System.out.println(message);  
  
        int expectedDistance = 10;  
        duck.swim(expectedDistance);  
        System.out.println("The expected distance is " + expectedDistance);  
    }  
}
```



## **Pitfall: Use of the Terms "Parameter" and "Argument"**

---

- ☐ Do not be surprised to find that people often use the terms parameter and argument interchangeably
- ☐ When you see these terms, you may have to determine their exact meaning from context



# The `this` Parameter

---

- ❑ All instance variables are understood to have `<the calling object>.` in front of them
- ❑ If an explicit name for the calling object is needed, the keyword `this` can be used
  - `myInstanceVariable` always means and is always interchangeable with `this.myInstanceVariable`



# The `this` Parameter

- ❑ **`this`** *must* be used if a parameter or other local variable with the same name is used in the method
  - Otherwise, all instances of the variable name will be interpreted as local

```
class A{  
    int someVariable = 10;  
    public void method1(int newVariable){  
        this.someVariable = newVariable  
    }  
}
```

↑                      ↑  
instance                local



# The **this** Parameter

---

- ❑ The **this** parameter is a kind of hidden parameter
- ❑ Even though it does not appear on the parameter list of a method, it is still a parameter
- ❑ When a method is invoked, the calling object is automatically plugged in for **this**



# Lab

---

```
public class Duck {  
  
    public boolean canfly = false;  
  
    public Duck(boolean canfly){  
        this.canfly = canfly;  
    }  
  
    public void quack(){  
        System.out.println("Quack!!");  
    }  
  
    public String eat(String food){  
        String message = "Thank you! The " + food + " is good!";  
        return message;  
    }  
  
    public void swim(int distance){  
        distance = distance - 1;  
        System.out.println("The distance of my swimming is " + distance);  
    }  
}
```



# Lab

---

```
public class Farm {

    public static void main(String[] args) {
        Duck duck = new Duck(true);

        boolean canTheDuckFly = duck.canfly;
        if(canTheDuckFly == true){
            System.out.println("The duck can fly");
        }

        duck.quack();

        String food = "Hamburger";
        String message = duck.eat(food);
        System.out.println(message);

        int expectedDistance = 10;
        duck.swim(expectedDistance);
        System.out.println("The expected distance is " + expectedDistance);
    }
}
```





# Encapsulation

---

- ❑ *Encapsulation* means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
  - Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface
  - In Java, hiding details is done by marking them **private**



# public and private Modifiers

---

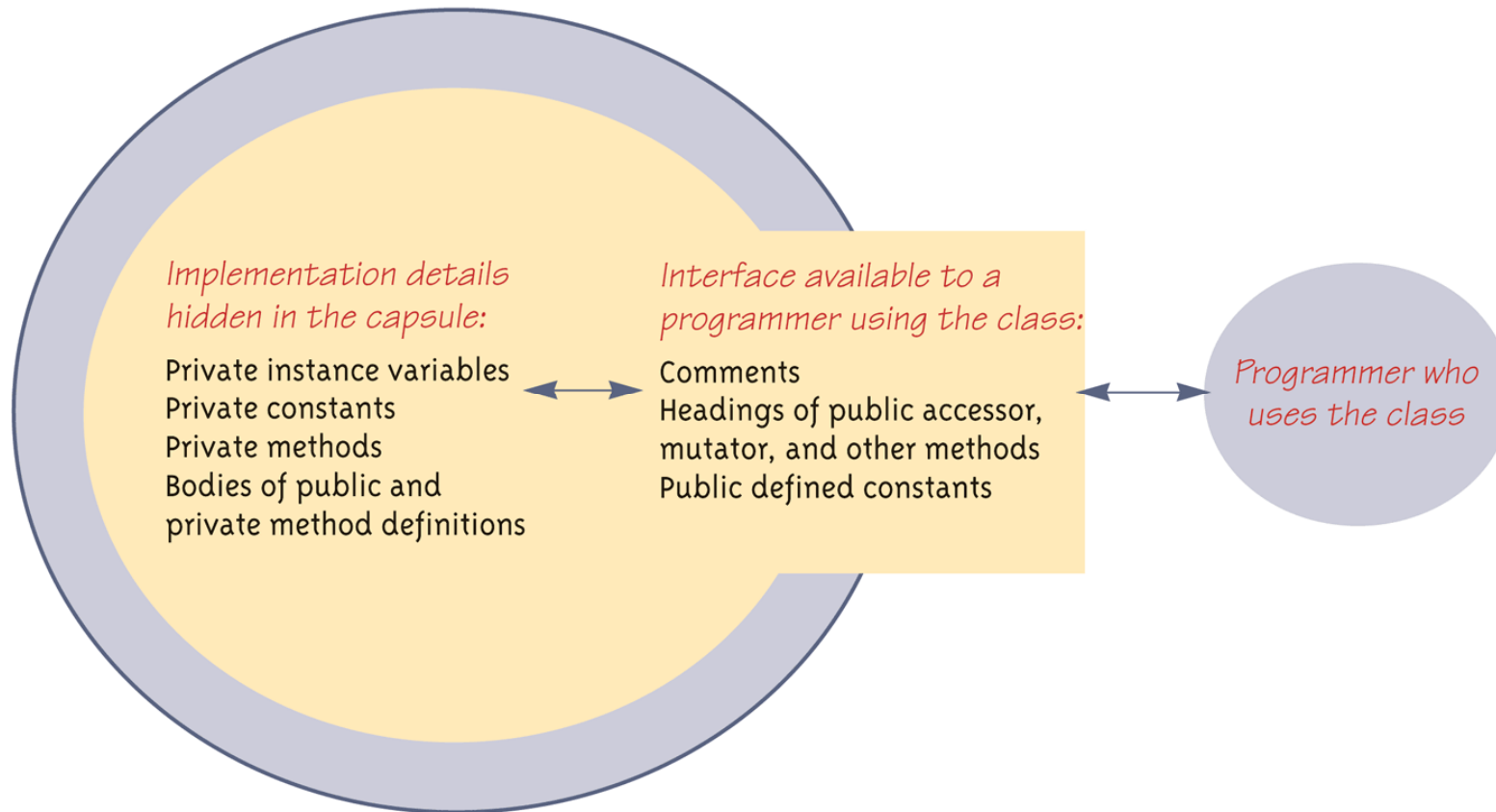
- ❑ The modifier **public** means that there are no restrictions on where an instance variable or method can be used
- ❑ The modifier **private** means that an instance variable or method cannot be accessed by name outside of the class
  - It is considered good programming practice to make **all** instance variables **private**
  - Most methods are **public**, and thus provide controlled access to the object
  - Usually, methods are **private** only if used as helping methods for other methods in the class



# Encapsulation

## Display 4.10 Encapsulation

*An encapsulated class*



*A class definition should have no public instance variables.*



# Lab

---

```
public class Duck {  
  
    private boolean canfly = false;  
  
    public boolean getCanfly(){  
        return canfly;  
    }  
  
    ...  
}
```



# Lab

---

```
public class Farm {  
  
    public static void main(String[] args) {  
        Duck duck = new Duck(true);  
  
        boolean canTheDuckFly = duck.getCanfly();  
        if(canTheDuckFly == true){  
            System.out.println("The duck can fly");  
        }  
  
        ...  
    }  
  
}
```



# Overloading

---

- ❑ *Overloading* is when two or more methods *in the same class* have the same method name
- ❑ To be valid, any two definitions of the method name must have different *signatures*
  - A signature consists of the name of a method together with its parameter list
  - Differing signatures must have different numbers and/or types of parameters



# Overloading and Automatic Type Conversion

---

- ❑ If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
- ❑ The interaction of overloading and automatic type conversion can have unintended results
- ❑ In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
  - Ambiguous method invocations will produce an error in Java



# Lab

---

```
public class Duck {  
    ...  
    public void quack(){  
        System.out.println("Quack!!");  
    }  
    public void quack(String sound){  
        System.out.println(sound);  
    }  
    ...  
}  
  
public class Farm {  
  
    public static void main(String[] args) {  
        Duck duck = new Duck(true);  
        ...  
        duck.quack();  
        duck.quack("Ga Ga Ga");  
    }  
}
```





## Pitfall: You Can Not Overload Based on the Type Returned

---

- ❑ The signature of a method only includes the method name and its parameter types
  - **The signature does not include the type returned**
- ❑ Java does not permit methods with the same name and different return types in the same class



## You Can Not Overload Operators in Java

---

- ❑ Although many programming languages, such as C++, allow you to overload operators (+, -, etc.), Java does not permit this
  - You may only use a method name and ordinary method syntax to carry out the operations you desire



# Constructors

---

- ❑ A *constructor* is a special kind of method that is designed to initialize the instance variables for an object:

```
public ClassName(anyParameters) {code}
```

- A constructor must have the same name as the class
- A constructor has no type returned, not even **void**
- Constructors are typically overloaded



# Constructors

---

- ❑ A constructor is called when an object of the class is created using **new**  
`ClassName objectName = new ClassName(anyArgs);`
  - The name of the constructor and its parenthesized list of arguments (if any) must follow the **new** operator
  - This is the **only** valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method
- ❑ If a constructor is invoked again (using **new**), the first object is discarded and an entirely new object is created
  - If you need to change the values of instance variables of the object, use mutator methods instead



## You Can Invoke Another Method in a Constructor

---

- ❑ The first action taken by a constructor is to create an object with instance variables
- ❑ Therefore, it is legal to invoke another method within the definition of a constructor, since it has the newly created object as its calling object
  - For example, mutator methods can be used to set the values of the instance variables
  - It is even possible for one constructor to invoke another



## A Constructor Has a **this** Parameter

---

- ☐ Like any ordinary method, every constructor has a **this** parameter
- ☐ The **this** parameter can be used explicitly, but is more often understood to be there than written down
- ☐ The first action taken by a constructor is to automatically create an object with instance variables
- ☐ Then within the definition of a constructor, the **this** parameter refers to the object created by the constructor



# Include a No-Argument Constructor

---

- ☐ If you do not include any constructors in your class, Java will automatically create a *default* or *no-argument* constructor that takes no arguments, performs no initializations, but allows the object to be created
- ☐ If you include even one constructor in your class, Java will not provide this default constructor
- ☐ If you include any constructors in your class, be sure to provide your own no-argument constructor as well



# Default Variable Initializations

---

- ❑ Instance variables are automatically initialized in Java
  - **boolean** types are initialized to **false**
  - Other primitives are initialized to the zero of their type
  - Class types are initialized to **null**
- ❑ However, it is a better practice to explicitly initialize instance variables in a constructor
- ❑ Note: Local variables are not automatically initialized





# The StringTokenizer Class

---

- ❑ The **StringTokenizer** class is used to recover the words or *tokens* in a multi-word **String**
  - You can use whitespace characters to separate each token, or you can specify the characters you wish to use as separators
  - In order to use the **StringTokenizer** class, be sure to include the following at the start of the file:  
**import java.util.StringTokenizer;**



# Some Methods in the StringTokenizer Class (Part 1 of 2)

## Display 4.17 Some Methods in the Class StringTokenizer

The class `StringTokenizer` is in the `java.util` package.

```
public StringTokenizer(String theString)
```

Constructor for a tokenizer that will use whitespace characters as separators when finding tokens in `theString`.

```
public StringTokenizer(String theString, String delimiters)
```

Constructor for a tokenizer that will use the characters in the string `delimiters` as separators when finding tokens in `theString`.

```
public boolean hasMoreTokens()
```

Tests whether there are more tokens available from this tokenizer's string. When used in conjunction with `nextToken`, it returns `true` as long as `nextToken` has not yet returned all the tokens in the string; returns `false` otherwise.

(continued)



## Some Methods in the StringTokenizer Class (Part 2 of 2)

### Display 4.17 Some Methods in the Class StringTokenizer

```
public String nextToken()
```

Returns the next token from this tokenizer's string. (Throws `NoSuchElementException` if there are no more tokens to return.)<sup>5</sup>

```
public String nextToken(String delimiters)
```

First changes the delimiter characters to those in the string `delimiters`. Then returns the next token from this tokenizer's string. After the invocation is completed, the delimiter characters are those in the string `delimiters`.

(Throws `NoSuchElementException` if there are no more tokens to return. Throws `NullPointerException` if `delimiters` is `null`.)<sup>5</sup>

```
public int countTokens()
```

Returns the number of tokens remaining to be returned by `nextToken`.



# Lab

---

```
import java.util.StringTokenizer;

public class StringTokenizerTest {

    public static void main(String[] args) {
        String in = "Hello,World,Java";
        StringTokenizer st = new StringTokenizer(in, ",");
        while(st.hasMoreTokens()) {
            String token = st.nextToken();
            System.out.println(token);
        }
    }
}
```



# Static Methods

---

- ❑ A *static method* is one that can be used without a calling object
- ❑ A static method still belongs to a class, and its definition is given inside the class definition
- ❑ When a static method is defined, the keyword **static** is placed in the method header

```
public static returnType myMethod(parameters)
{ . . . }
```
- ❑ Static methods are invoked using the class name in place of a calling object

```
returnValue = MyClass.myMethod(arguments);
```



# Lab

---

```
public class StaticTest {  
  
    public static void main(String[] args) {  
  
        int sum = Tool.add(1,1);  
  
        System.out.println(sum);  
    }  
}  
  
public class Tool {  
  
    public static int add(int a, int b){  
        return a+b;  
    }  
}
```



## Pitfall: Invoking a Nonstatic Method Within a Static Method

---

- ❑ A static method cannot refer to an instance variable of the class, and it cannot invoke a nonstatic method of the class
  - A static method has no **this**, so it cannot use an instance variable or method that has an implicit or explicit **this** for a calling object
  - A static method can invoke another static method, however



# Lab

---

```
public class StaticTest {  
  
    public static void main(String[] args) {  
  
        int sum = add(1,1);  
        System.out.println(sum);  
  
        StaticTest st = new StaticTest();  
        sum = st.add2(2,2);           //Cannot be: sum = add2(2,2);  
    }  
  
    public static int add(int a, int b){  
        return a+b;  
    }  
  
    public int add2(int a, int b){  
        return a+b;  
    }  
  
}
```





# Static Variables

---

- ❑ A *static variable* is a variable that belongs to the class as a whole, and not just to one object
  - There is only one copy of a static variable per class, unlike instance variables where each object has its own copy
- ❑ All objects of the class can read and change a static variable
- ❑ Although a static method cannot access an instance variable, a static method can access a static variable
- ❑ A static variable is declared like an instance variable, with the addition of the modifier **static**  
`private static int myStaticVariable;`



# Static Variables

---

- ❑ Static variables can be declared and initialized at the same time

```
private static int myStaticVariable = 0;
```

- ❑ If not explicitly initialized, a static variable will be automatically initialized to a default value
  - **boolean** static variables are initialized to **false**
  - Other primitive types static variables are initialized to the zero of their type
  - Class type static variables are initialized to **null**
- ❑ It is always preferable to explicitly initialize static variables rather than rely on the default initialization



# Static Variables

---

- ❑ A static variable should always be defined private, unless it is also a defined constant

- The value of a static defined constant cannot be altered, therefore it is safe to make it **public**
- In addition to **static**, the declaration for a static defined constant must include the modifier **final**, which indicates that its value cannot be changed

```
public static final int BIRTH_YEAR = 1954;
```

- ❑ When referring to such a defined constant outside its class, use the name of its class in place of a calling object

```
int year = MyClass.BIRTH_YEAR;
```



# Lab

---

```
public class StaticTest {  
  
    public static int port = 80;  
  
    public static void main(String[] args) {  
        StaticTest obj1 = new StaticTest();  
        StaticTest obj2 = new StaticTest();  
  
        System.out.println(StaticTest.port);  
        System.out.println(obj1.port);  
        System.out.println(obj2.port);  
  
        StaticTest.port = 1234;  
        System.out.println(obj1.port);  
  
        obj2.port = 5678;  
        System.out.println(obj1.port);  
    }  
}
```



# The Math Class

---

- ❑ The **Math** class provides a number of standard mathematical methods
  - It is found in the **java.lang** package, so it does not require an **import** statement
  - All of its methods and data are static, therefore they are invoked with the class name **Math** instead of a calling object
  - The **Math** class has two predefined constants, **E** ( $e$ , the base of the natural logarithm system) and **PI** ( $\pi$ , 3.1415 ...)

```
area = Math.PI * radius * radius;
```



# Some Methods in the Class Math

## (Part 1 of 5)

---

### Display 5.6 Some Methods in the Class Math

---

The Math class is in the `java.lang` package, so it requires no `import` statement.

```
public static double pow(double base, double exponent)
```

Returns base to the power exponent.

#### **EXAMPLE**

`Math.pow(2.0, 3.0)` returns `8.0`.

(continued)



# Some Methods in the Class Math

## (Part 2 of 5)

### Display 5.6 Some Methods in the Class Math

```
public static double abs(double argument)
public static float abs(float argument)
public static long abs(long argument)
public static int abs(int argument)
```

Returns the absolute value of the argument. (The method name `abs` is overloaded to produce four similar methods.)

#### EXAMPLE

`Math.abs(-6)` and `Math.abs(6)` both return 6. `Math.abs(-5.5)` and `Math.abs(5.5)` both return 5.5.

```
public static double min(double n1, double n2)
public static float min(float n1, float n2)
public static long min(long n1, long n2)
public static int min(int n1, int n2)
```

Returns the minimum of the arguments `n1` and `n2`. (The method name `min` is overloaded to produce four similar methods.)

#### EXAMPLE

`Math.min(3, 2)` returns 2.

(continued)



# Some Methods in the Class Math

## (Part 3 of 5)

### Display 5.6 Some Methods in the Class Math

```
public static double max(double n1, double n2)
public static float max(float n1, float n2)
public static long max(long n1, long n2)
public static int max(int n1, int n2)
```

Returns the maximum of the arguments n1 and n2. (The method name max is overloaded to produce four similar methods.)

#### EXAMPLE

Math.max(3, 2) returns 3.

```
public static long round(double argument)
public static int round(float argument)
```

Rounds its argument.

#### EXAMPLE

Math.round(3.2) returns 3; Math.round(3.6) returns 4.

(continued)





## Some Methods in the Class Math (Part 4 of 5)

---

### Display 5.6 Some Methods in the Class Math

---

```
public static double ceil(double argument)
```

Returns the smallest whole number greater than or equal to the argument.

#### **EXAMPLE**

`Math.ceil(3.2)` and `Math.ceil(3.9)` both return `4.0`.

(continued)



# Some Methods in the Class Math

## (Part 5 of 5)

---

### Display 5.6 Some Methods in the Class Math

---

```
public static double floor(double argument)
```

Returns the largest whole number less than or equal to the argument.

#### **EXAMPLE**

`Math.floor(3.2)` and `Math.floor(3.9)` both return `3.0`.

```
public static double sqrt(double argument)
```

Returns the square root of its argument.

#### **EXAMPLE**

`Math.sqrt(4)` returns `2.0`.



# Random Numbers

---

- ❑ The **Math** class also provides a facility to generate pseudo-random numbers

`public static double random()`

- A pseudo-random number appears random but is really generated by a deterministic function

- There is also a more flexible class named **Random**

- ❑ Sample use: `double num = Math.random();`

- ❑ Returns a pseudo-random number greater than or equal to 0.0 and less than 1.0



# Lab

```
public class MathExample {  
  
    public static void main(String[] args){  
        int i = 7;  
        int j = -9;  
        double x = 72.3;  
        double y = 0.34;  
  
        System.out.println("i is " + i);  
        System.out.println("j is " + j);  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
  
        System.out.println("/" + i + "/" + " is " + Math.abs(i));  
        System.out.println("/" + j + "/" + " is " + Math.abs(j));  
        System.out.println("/" + x + "/" + " is " + Math.abs(x));  
        System.out.println("/" + y + "/" + " is " + Math.abs(y));  
  
        System.out.println(x + " is approximately " + Math.round(x));  
        System.out.println(y + " is approximately " + Math.round(y));  
    }  
}
```



# Lab

```
System.out.println("The ceiling of " + i + " is " + Math.ceil(i));
System.out.println("The ceiling of " + j + " is " + Math.ceil(j));
System.out.println("The ceiling of " + x + " is " + Math.ceil(x));
System.out.println("The ceiling of " + y + " is " + Math.ceil(y));

System.out.println("min(" + i + ", " + j + ") is " + Math.min(i,j));
System.out.println("min(" + x + ", " + y + ") is " + Math.min(x,y));
System.out.println("min(" + i + ", " + x + ") is " + Math.min(i,x));
System.out.println("min(" + y + ", " + j + ") is " + Math.min(y,j));

System.out.println("max(" + i + ", " + j + ") is " + Math.max(i,j));
System.out.println("max(" + x + ", " + y + ") is " + Math.max(x,y));
System.out.println("max(" + i + ", " + x + ") is " + Math.max(i,x));
System.out.println("max(" + y + ", " + j + ") is " + Math.max(y,j));

System.out.println("Pi is " + Math.PI);
System.out.println("e is " + Math.E);

System.out.println("pow(2.0, 2.0) is " + Math.pow(2.0,2.0));
System.out.println("pow(10.0, 3.5) is " + Math.pow(10.0,3.5));
System.out.println("pow(8, -1) is " + Math.pow(8, -1));

System.out.println("Here's one random number: " + Math.random());
System.out.println("Here's another random number: " + Math.random());
}
}
```



# Wrapper Classes

---

- ❑ *Wrapper classes* provide a class type corresponding to each of the primitive types
  - This makes it possible to have class types that behave somewhat like primitive types
  - The wrapper classes for the primitive types **byte**, **short**, **long**, **float**, **double**, and **char** are (in order) **Byte**, **Short**, **Long**, **Float**, **Double**, and **Character**
- ❑ Wrapper classes also contain a number of useful predefined constants and static methods



# Wrapper Classes

---

- ❑ *Boxing*: the process of going from a value of a primitive type to an object of its wrapper class
  - To convert a primitive value to an "equivalent" class type value, create an object of the corresponding wrapper class using the primitive value as an argument
  - The new object will contain an instance variable that stores a copy of the primitive value
  - Unlike most other classes, a wrapper class does not have a no-argument constructor

```
Integer integerObject = new Integer(42);
```



# Wrapper Classes

---

- ❑ *Unboxing*: the process of going from an object of a wrapper class to the corresponding value of a primitive type
  - The methods for converting an object from the wrapper classes **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, and **Character** to their corresponding primitive type are (in order) **byteValue**, **shortValue**, **intValue**, **longValue**, **floatValue**, **doubleValue**, and **charValue**
  - None of these methods take an argument  
`int i = integerObject.intValue();`





# Automatic Boxing and Unboxing

---

- ❑ Starting with version 5.0, Java can automatically do boxing and unboxing
- ❑ Instead of creating a wrapper class object using the **new** operation (as shown before), it can be done as an automatic type cast:

```
Integer integerObject = 42;
```

- ❑ Instead of having to invoke the appropriate method (such as **intValue**, **doubleValue**, **charValue**, etc.) in order to convert from an object of a wrapper class to a value of its associated primitive type, the primitive value can be recovered automatically

```
int i = integerObject;
```



# Constants and Static Methods in Wrapper Classes

---

- ❑ Wrapper classes have static methods that convert a correctly formed string representation of a number to the number of a given type
  - The methods `Integer.parseInt`, `Long.parseLong`, `Float.parseFloat`, and `Double.parseDouble` do this for the primitive types (in order) `int`, `long`, `float`, and `double`
- ❑ Wrapper classes also have static methods that convert from a numeric value to a string representation of the value
  - For example, the expression  
`Double.toString(123.99);`  
returns the string value `"123.99"`
- ❑ The `Character` class contains a number of static methods that are useful for string processing



# Lab

---

```
public class WrapperClassTest {  
  
    public static void main(String[] args) {  
        int k = 100;  
        Integer it1 = new Integer(k);  
        int m = it1.intValue();  
  
        System.out.println(m*k);  
    }  
}
```



# Lab

---

```
public class WrapperClassTest {  
  
    public static void main(String[] args) {  
        Integer it1 = new Integer(100);  
        System.out.println(it1);  
    }  
}
```



# Lab

---

```
public class WrapperClassTest {  
  
    public static void main(String[] args) {  
  
        double pi = Double.parseDouble("3.14");  
        System.out.println(pi);  
  
        Double d = new Double("1.5");  
        String str = d.toString();  
        System.out.println(str);  
    }  
}
```



# Lab

---

```
import java.util.Scanner;
import java.util.StringTokenizer;

public class Parser {
    public static void main(String args[]) {
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter a sentence and I'll display each
word you entered: ");
        String sentence = keyboard.nextLine();

        // Parse the string into tokens and echo back to the user
        StringTokenizer tk = new StringTokenizer(sentence, " ");
        System.out.println("Here are the tokens: ");
        while (tk.hasMoreTokens()) {
            System.out.println(tk.nextToken());
        }
    }
}
```



# Lab

---

```
public class Student {
    private String name;
    private double gpa;

    /** Constructors */
    public Student() {
        name = null;
        gpa = 0.0;
    }
    public Student(String n, double g) {
        name = n;
        gpa = g;
    }
    /** Accessor methods */
    public String getName() {
        return name;
    }
    public double getGPA() {
        return gpa;
    }
    /** Mutator methods */
    public void setName(String n) {
        name = n;
    }
    public void setGPA(double g) {
        if ((g >= 0) && (g <= 4))
            gpa = g;
    }
}
```



# Lab

```
/** Facilitator methods */
public String toString() {
    return (name + ":" + gpa);
}
public boolean equals(Student s) {
    return (name.equalsIgnoreCase(s.name));
}
public static void main(String[] args){
    Student student1 = new Student("Mike", 90);
    student1.setGPA(92);

    System.out.println(student1.getName());
    System.out.println(student1.getGPA());
    System.out.println(student1.toString());

    Student student2 = new Student("Mary", 90);
    if(student2.equals(student1)){
        System.out.println("student1 is student2!");
    }else{
        System.out.println("student1 is not student2!");
    }
}
}
```





## Reference

---

- ❑ “Absolute Java”. Walter Savitch and Kenrick Mock. Addison-Wesley; 5 edition. 2012
- ❑ “Java How to Program”. Paul Deitel and Harvey Deitel. Prentice Hall; 9 edition. 2011.
- ❑ “A Programmers Guide To Java SCJP Certification: A Comprehensive Primer 3rd Edition”. Khalid Mughal, Rolf Rasmussen. Addison-Wesley Professional. 2008