

2 Object Oriented Programming

Objectives

- ▶ Define modeling concepts: abstraction, encapsulation, and packages
- ▶ Discuss why you can reuse Java technology application code
- ▶ Define class, member, attribute, method, constructor, and package
- ▶ Use the access modifiers private, and public as appropriate for the guidelines of the encapsulation
- ▶ Invoke a method on a particular object

Objectives

- ▶ In a Java program, identify the following:
 - The package statement
 - The import statements
 - Classes, methods, and attributes
 - Constructors
- ▶ Use the Java technology application programming interface (API) online documentation

Software Engineering

Toolkits / Frameworks / Object APIs (1990s – up)

Java 2 SDK AWT / Swing Jini Java Beans JDBC

Object-Oriented Languages (1980s – up)

SELF Smalltalk Common Lisp Object System Eiffel C++ Java

Libraries / Functional APIs (1960s – early 1980s)

NASTRAN TCP/IP ISAM X-Windows OpenLook

High-Level Languages (1950s –up)

Fortran LISP C COBOL

Operating Systems (1960s – up)

OS/360 UNIX MacOS MS-Windows

Machine Code (late 1940s – up)

The Analysis and Design Phase

- ▶ Analysis describes what the system needs to do:
 - Modeling the real-world: actors and activities, objects, and behaviors
- ▶ Design describes how the system does it:
 - Modeling the relationships and interactions between objects and actors in the system
 - Finding useful abstractions to help simplify the problem or solution

Abstraction

- ▶ Functions – Write an algorithm once to be used in many situations
 - Objects – Group a related set of attributes and behaviors into a class
 - Frameworks and APIs – Large groups of objects that support a complex activity:
 - Frameworks can be used “as is” or be modified to extend the basic behavior

Classes as Blueprints for Objects

- ▶ In manufacturing, a blueprint describes a device from which many physical devices are constructed
- ▶ In software, a class is a description of an object:
 - A class describes the data that each object includes
 - A class describes the behaviors that each object exhibits
- ▶ In Java technology, classes support three key features of object-oriented programming (OOP):
 - Encapsulation
 - Inheritance
 - Polymorphism

Declaring Java Technology Classes

- Basic syntax of a Java class:

```
< modifiers> class < class_name> {  
    [< attribute_declarations>]  
    [< constructor_declarations>]  
    [< method_declarations>]  
}
```

- Example:

```
public class Vehicle {  
    private double maxLoad;  
    public void setMaxLoad(double value{  
        maxLoad = value;  
    }  
}
```

Declaring Attributes

- ▶ Basic syntax of an attribute:

< modifiers> <type> <name>;

- ▶ Examples:

```
public class Foo {  
    private int x;  
    private float y = 10000.0F;  
    private String name = "Bates Motel";  
}
```

Declaring Methods

- ▶ Basic syntax of a method:

```
<modifiers> <return_type> <name>
    ([< argument_list>]) {
        [< statements>]
    }
```

- ▶ Examples:

```
public class Dog {
    private int weight;
    public int getWeight() {
        return weight;
    }
    public void setWeight(int newWeight) {
        weight = newWeight;
    }
}
```

Accessing Object Members

- ▶ The “dot” notation:
`<object>.<member>`
- ▶ This is used to access object members including attributes and methods
- ▶ Examples:
`d.setWeight(42);`
`d.weight = 42; // only permissible`
`//if weight is public`

Information Hiding

► The Problem:

MyDate

(from Logical View)

day

month

year

Client code has direct access to internal data:

```
MyDate d = new MyDate();
```

```
d.day = 32;
```

```
// invalid day
```

```
d.month = 2; d.day = 30;
```

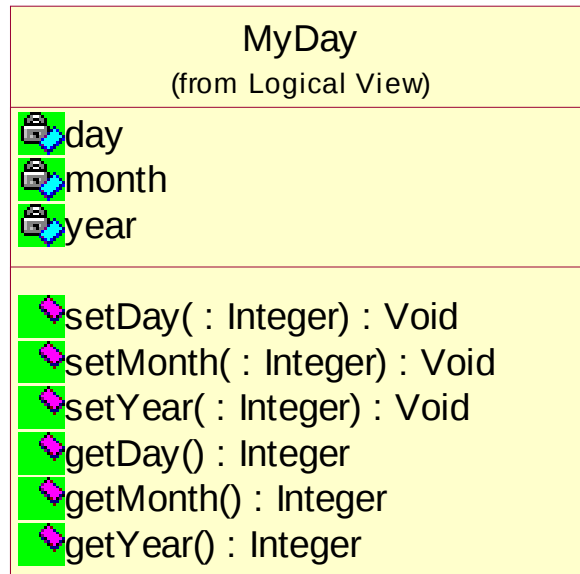
```
// plausible but wrong
```

```
d.day = d.day + 1;
```

```
// no check for wrap around
```

Information Hiding

► The Solution:



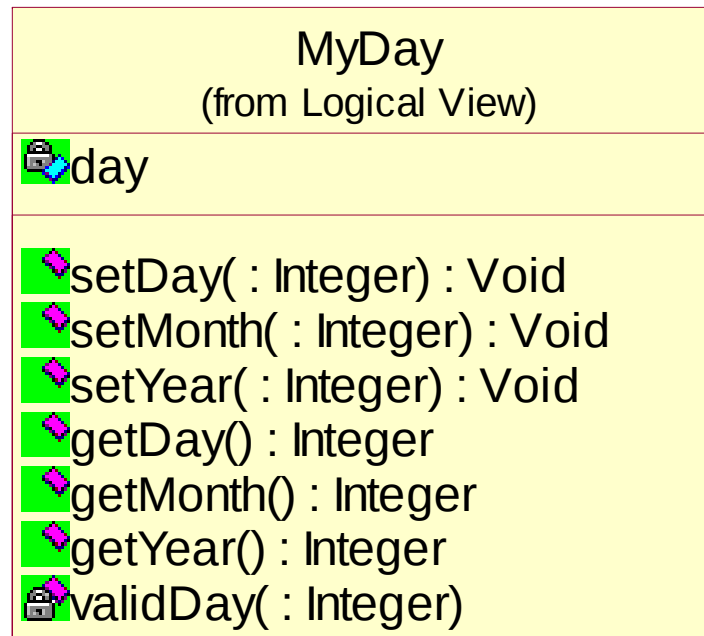
verify days in month

Client code must use setters/getters to access internal data:

```
MyDate d = new MyDate();  
d.setDay(32);  
// invalid day, returns false  
d.setMonth(2);  
d.setDay(30);  
// plausible but wrong  
d.setDay(d.getDay() + 1);
```

Encapsulation

- Hides the implementation details of a class
 - Forces the user to use an interface to access data
 - Makes the code more maintainable



Declaring Constructors

```
1 public class Dog {  
2     private int weight;  
3  
4     public Dog() {  
5         weight = 42;  
6     }  
7  
8     public int getWeight() {  
9         return weight;  
10    }  
11    public void setWeight(int newWeight) {  
12        weight = newWeight;  
13    }  
14}
```

The Default Constructor

- ▶ There is always at least one constructor in every class.
- ▶ If the writer does not supply any constructors, the default constructor is present automatically:
 - The default constructor takes no arguments
 - The default constructor has no body
- ▶ Enables you to create object instances with new Xxx() without having to write a constructor.

Source File Layout

- ▶ Basic syntax of a Java source file:

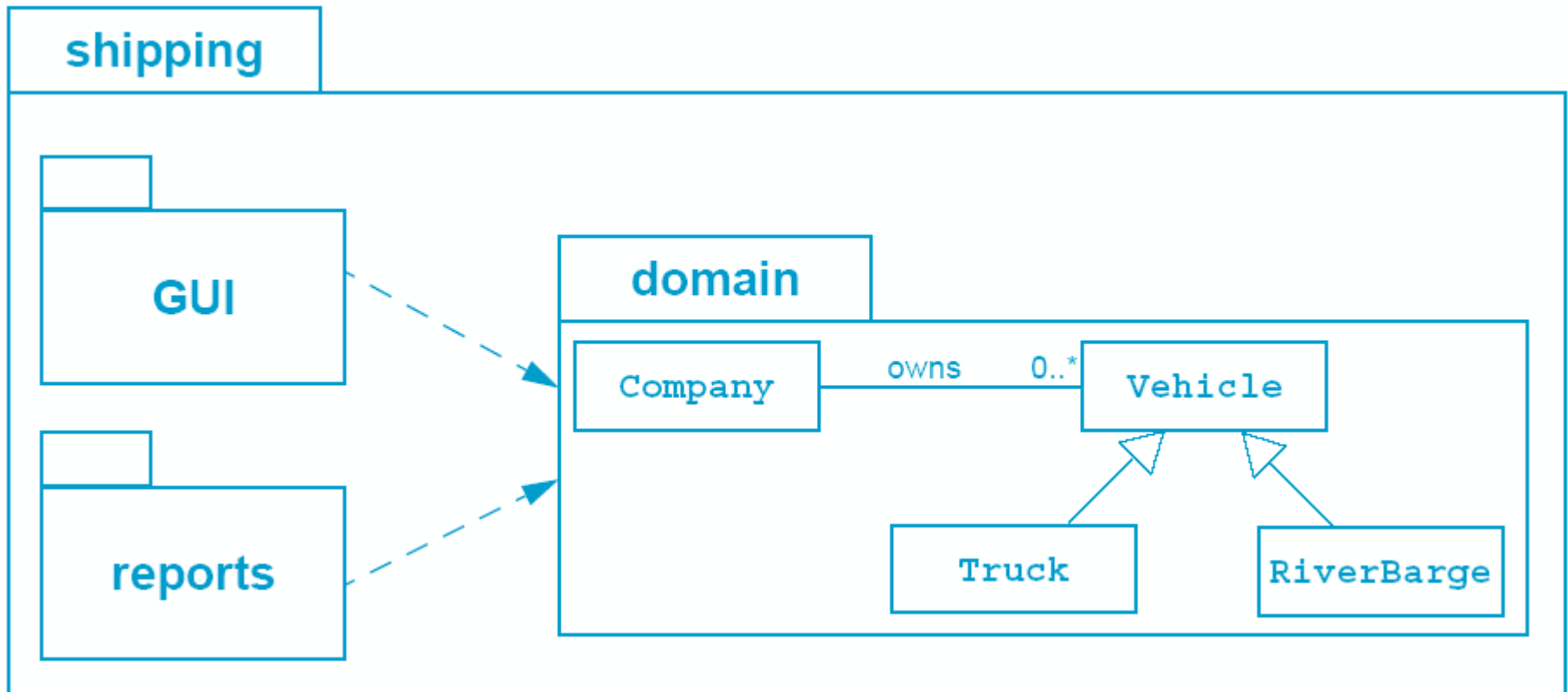
```
[< package_declaration>]  
[< import_declarations>]  
< class_declaration>+
```

- ▶ Example, the VehicleCapacityReport.java file:

```
package shipping.reports;  
import shipping.domain.*;  
import java.util.List;  
import java.io.*;  
public class VehicleCapacityReport {  
    private List vehicles;  
    public void generateReport(Writer output)  
    {...}  
}
```

Software Packages

- ▶ Packages help manage large software systems.
- ▶ Packages can contain classes and sub-packages.



The package Statement

- ▶ Basic syntax of the package statement:

```
package < top_pkg_name>[.< sub_pkg_name>]*;
```

- ▶ Example:

```
package shipping.reports;
```

- ▶ Specify the package declaration at the beginning of the source file.
- ▶ Only one package declaration per source file.
- ▶ If no package is declared, then the class “belongs” to the default package.
- ▶ Package names must be hierarchical and separated by dots.

The `import` Statement

- ▶ Basic syntax of the `import` statement:

```
import  
<pkg_name>[.<sub_pkg_name>].<class_name>;
```

- ▶ OR

```
import <pkg_name>[.< sub_pkg_name>].*;
```

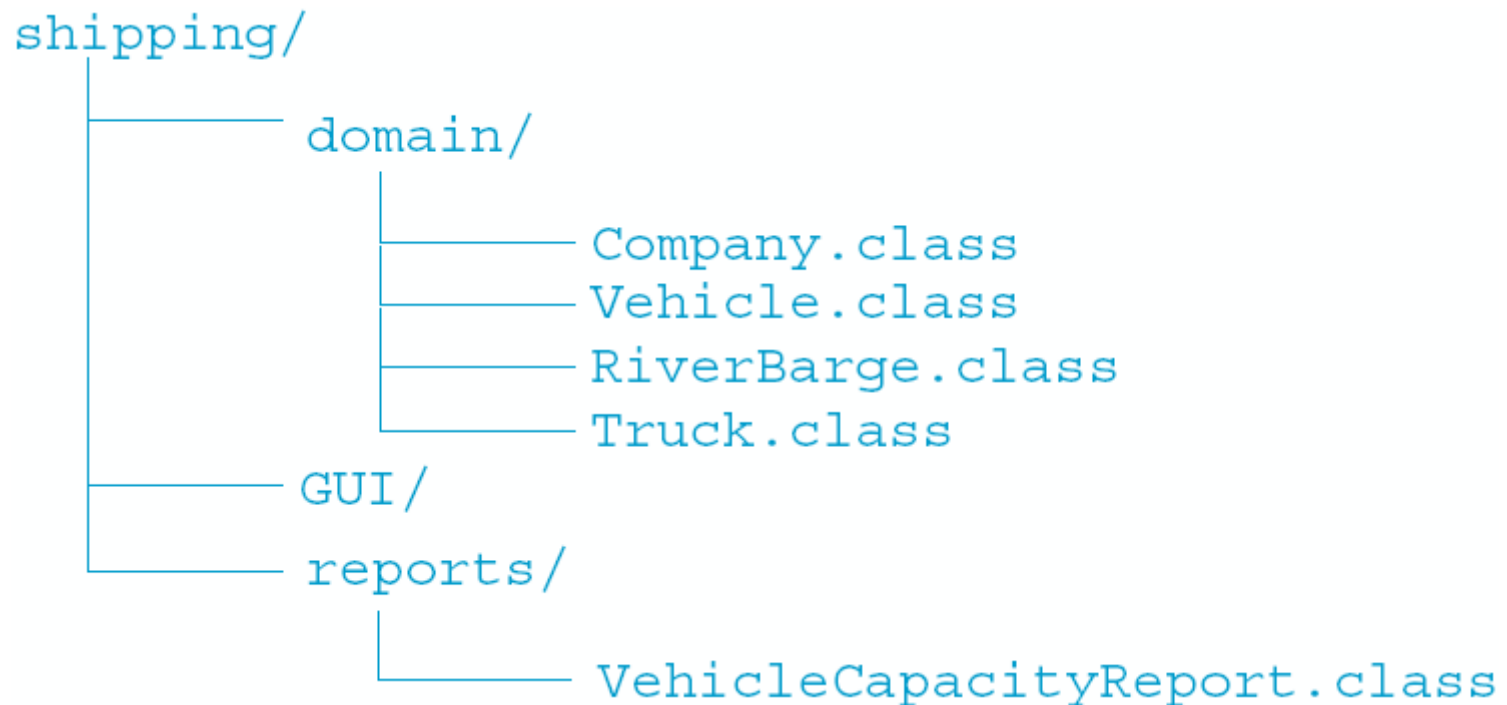
- ▶ Examples:

```
import shipping.domain.*;  
import java.util.List;  
import java.io.*;
```

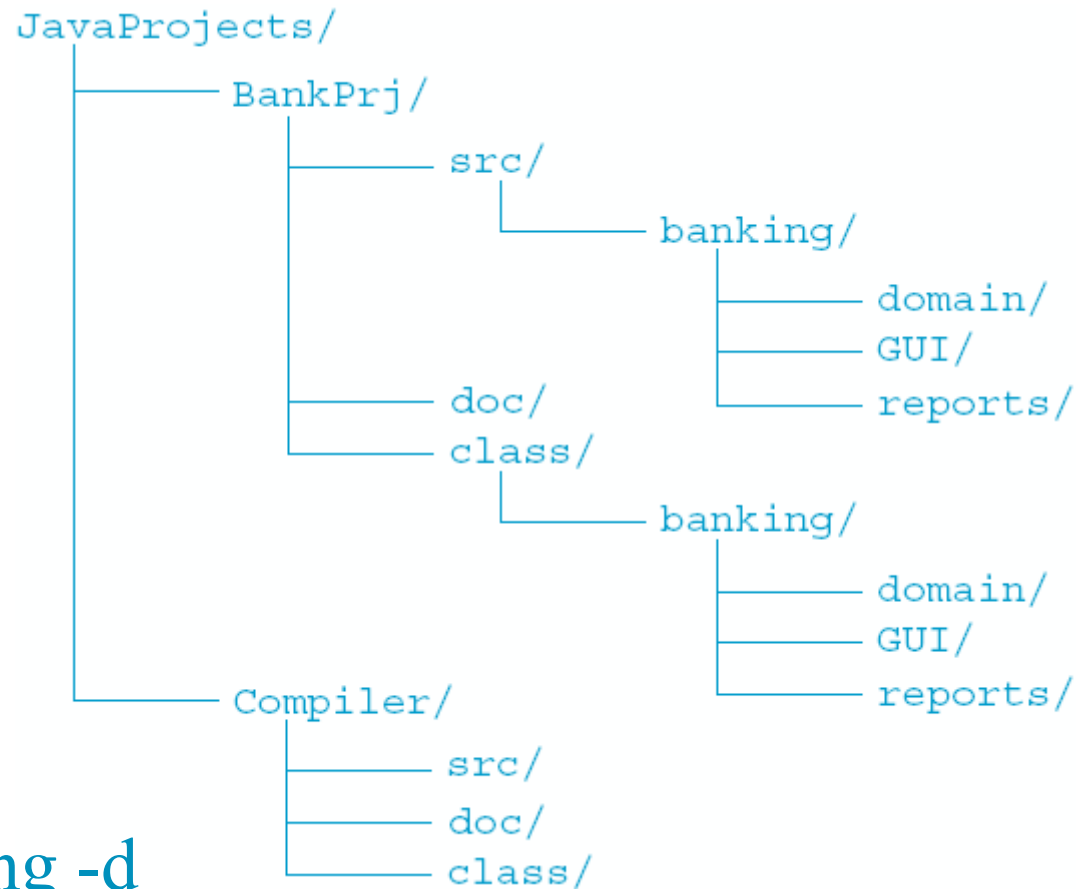
- ▶ Precedes all class declarations
- ▶ Tells the compiler where to find classes to use

Directory Layout and Packages

- ▶ Packages are stored in the directory tree containing the package name.
- ▶ Example, the “shipping” application packages:



Development



► Compiling using -d
`cd JavaProjects/BankPrj/src`
`javac -d ../class banking/domain/*.java`

Using the Java API Documentation

- ▶ A set of Hypertext Markup Language (HTML) files provides information about the API.
- ▶ One package contains hyperlinks to information on all of the classes.
- ▶ A class document includes the class hierarchy, a description of the class, a list of member variables, a list of constructors, and so on.

Example API Documentation Page

The screenshot shows a Microsoft Internet Explorer window displaying the Java 2 Platform API Specification for Standard Edition, v1.4.0. The browser's address bar shows the local file path: C:\jdk1.5.0\docs\api\index.html. The page has a navigation bar with links: Overview, Package, Class, Use, Tree, Deprecated, Index, and Help. The main content area is titled "Java™ 2 Platform, Standard Edition, v 1.4.0 API Specification" and includes a brief description of the document. On the left side, there are two panels: "All Classes" listing various abstract classes like AbstractAction, AbstractBorder, etc., and "Packages" listing sub-packages like java.applet, java.awt, etc. The right side of the page features a table titled "Java 2 Platform Packages" which lists the packages and their descriptions.

Java 2 Platform Packages	
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing beans -- components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.

Declaring Java Technology Classes

```
public class Circle {
```

```
    private double x, y, r; // The center and the radius of the circle
```

```
    public Circle ( double x, double y, double r ) {
```

```
        this.x = x; this.y = y; this.r = r;
```

```
    }
```

```
    public void setCenter(double a,double b){ x=a ; y=b ; }
```

```
    public void setRadius(double R){ r=R; }
```

```
    public double circumference( ) { return 2 * 3.14159 * r; }
```

```
    public double area( ) { return 3.14159 * r*r; }
```

```
}
```

Declaring Attributes

```
public class Circle {
```

```
    private double x, y, r; // The center and the radius of the circle
```

```
    public Circle ( double x, double y, double r ) {
```

```
        this.x = x; this.y = y; this.r = r;
```

```
    }
```

```
    public void setCenter(double a,double b){ x=a ; y=b ; }
```

```
    public void setRadius(double R){ r=R; }
```

```
    public double circumference( ) { return 2 * 3.14159 * r; }
```

```
    public double area( ) { return 3.14159 * r*r; }
```

```
}
```

Declaring Methods

```
public class Circle {
```

```
    private double x, y, r; // The center and the radius of the circle
```

```
    public Circle ( double x, double y, double r ) {
```

```
        this.x = x; this.y = y; this.r = r;
```

```
    }
```

```
    public void setCenter(double a,double b){ x=a ; y=b ; }
```

```
    public void setRadius(double R){ r=R; }
```

```
    public double circumference( ) { return 2 * 3.14159 * r; }
```

```
    public double area( ) { return 3.14159 * r*r; }
```

```
}
```

Accessing Object Members

- ▶ The “dot” notation : **<object>.<member>**
- ▶ This is used to access object members including attributes and methods
- ▶ Examples:

```
c.setCenter(8.7, 23.5) ;
```

```
c.setRadius(3.14) ;
```

```
double a = c.area() ;
```

Information Hiding

```
Circle c ;
```

```
...
```

```
c = new Circle();
```

```
...
```

```
c.x = 2.0;
```

```
c.y = 2.0;
```

```
c.r = 1.0;
```

Declaring Constructors

```
public class Circle {
```

```
    private double x, y, r; // The center and the radius of the circle
```

```
    public Circle ( double x, double y, double r ) {
```

```
        this.x = x; this.y = y; this.r = r;
```

```
    }
```

```
    public void setCenter(double a,double b){ x=a ; y=b ; }
```

```
    public void setRadius(double R){ r=R; }
```

```
    public double circumference( ) { return 2 * 3.14159 * r; }
```

```
    public double area( ) { return 3.14159 * r*r; }
```

```
}
```

The Default Constructor

- ▶ There is always at least one constructor in every class,
- ▶ If the writer does not supply any constructors, the default constructor is present automatically:
 - The default constructor takes no arguments,
 - The default constructor has no body.
- ▶ Enables you to create object instances with
`new ClassName()` without having to write a constructor.

2# Object-Oriented Programming

- ▶ Exercise-1: “Java 2 Platform API Specification”
- ▶ Exercise-2: “Encapsulation”
- ▶ Exercise-3: “Creating a Simple Bank Package”

