Objects and Classes

• Defining Classes for Objects

```java
class Circle {
    /** The radius of this circle */
    double radius = 1;

    /** Construct a circle object */
    Circle() {
    }

    /** Construct a circle object */
    Circle(double newRadius) {
        radius = newRadius;
    }

    /** Return the area of this circle */
    double getArea() {
        return radius * radius * Math.PI;
    }

    /** Return the perimeter of this circle */
    double getPerimeter() {
        return 2 * radius * Math.PI;
    }

    /** Set new radius for this circle */
    double setRadius(double newRadius) {
        radius = newRadius;
    }
}
```
// File TestSimpleCircle.java

class SimpleCircle {
    ...
}

class TestSimpleCircle {
    ...
}
Constructing Objects Using Constructors

Constructors are a special kind of method. They have three peculiarities:

- A constructor must have the same name as the class itself.
- Constructors do not have a return type—not even `void`.
- Constructors are invoked using the `new` operator when an object is created. Constructors play the role of initializing objects.

The constructor has exactly the same name as its defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values.

It is a common mistake to put the `void` keyword in front of a constructor. For example,

```java
public void Circle() {
}
```

In this case, `Circle()` is a method, not a constructor.

 Constructors are used to construct objects. To construct an object from a class, invoke a constructor of the class using the `new` operator, as follows:

```java
new ClassName(arguments);
```
Reference Variables and Reference Types

Objects are accessed via the object’s *reference variables*, which contain references to the objects. Such variables are declared using the following syntax:

```
ClassName objectRefVar;
```

A class is essentially a programmer-defined type. A class is a *reference type*, which means that a variable of the class type can reference an instance of the class. The following statement declares the variable `myCircle` to be of the `Circle` type:

```
Circle myCircle;
```

The variable `myCircle` can reference a `Circle` object. The next statement creates an object and assigns its reference to `myCircle`:

```
myCircle = new Circle();
```

You can write a single statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable with the following syntax:

```
ClassName objectRefVar = new ClassName();
```

Here is an example:

```
Circle myCircle = new Circle();
```

The variable `myCircle` holds a reference to a `Circle` object.
Accessing an Object’s Data and Methods

- `objectRefVar.dataField` references a data field in the object.
- `objectRefVar.method(arguments)` invokes a method on the object.
Reference Data Fields and the `null` Value

If a data field of a reference type does not reference any object, the data field holds a special Java value, `null`. `null` is a literal just like `true` and `false`. While `true` and `false` are Boolean literals, `null` is a literal for a reference type.

The default value of a data field is `null` for a reference type, `0` for a numeric type, `false` for a `boolean` type, and `\u0000` for a `char` type. However, Java assigns no default value to a local variable inside a method. The following code displays the default values of the data fields `name`, `age`, `isScienceMajor`, and `gender` for a `Student` object:
Differences between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. For a variable of a primitive type, the value is of the primitive type. For a variable of a reference type, the value is a reference to where an object is located. For example, as shown in Figure 8.7, the value of `int` variable `i` is `int` value `1`, and the value of `Circle` object `c` holds a reference to where the contents of the `Circle` object are stored in memory.

When you assign one variable to another, the other variable is set to the same value. For a variable of a primitive type, the real value of one variable is assigned to the other variable. For a variable of a reference type, the reference of one variable is assigned to the other variable. As shown in Figure 8.8, the assignment statement `i = j` copies the contents of `j` into `i` for

![Diagram](image)

**Figure 8.7** A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.

![Diagram](image)

**Figure 8.8** Primitive variable `j` is copied to variable `i`.
primitive variables. As shown in Figure 8.9, the assignment statement \( c1 = c2 \) copies the reference of \( c2 \) into \( c1 \) for reference variables. After the assignment, variables \( c1 \) and \( c2 \) refer to the same object.

**Figure 8.9** Reference variable \( c2 \) is copied to variable \( c1 \).
Using Classes from the Java Library

• The Date Class

```java
java.util.Date

+Date()
+Date(elapseTime: long)
+toString(): String
+getTime(): long
+setTime(elapseTime: long): void
```

- Constructs a Date object for the current time.
- Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
- Returns a string representing the date and time.
- Returns the number of milliseconds since January 1, 1970, GMT.
- Sets a new elapse time in the object.
The Random Class

```java
java.util.Random

+Random()
  Constructs a Random object with the current time as its seed.

+Random(seed: long)
  Constructs a Random object with a specified seed.

+nextInt(): int
  Returns a random int value.

+nextInt(n: int): int
  Returns a random int value between 0 and n (excluding n).

+nextLong(): long
  Returns a random long value.

+nextDouble(): double
  Returns a random double value between 0.0 and 1.0 (excluding 1.0).

+nextFloat(): float
  Returns a random float value between 0.0F and 1.0F (excluding 1.0F).

+nextBoolean(): boolean
  Returns a random boolean value.
```
Static Variables, Constants, and Methods

• A static variable is shared by all objects of the class. A static method cannot access instance members of the class.

• Also called class variables or class methods.
static int numberOfObjects;

static int getNumberOfObjects() {
    return numberOfObjects;
}

UML Notation:
underline: static variables or methods

Tip
Use ClassName.methodName(arguments) to invoke a static method and ClassName.staticVariable to access a static variable. This improves readability, because other programmers can easily recognize the static method and data in the class.
An instance method can invoke an instance or static method and access an instance or static data field. A static method can invoke a static method and access a static data field. However, a static method cannot invoke an instance method or access an instance data field, since static methods and static data fields don’t belong to a particular object. The relationship between static and instance members is summarized in the following diagram:
**Design Guide**

How do you decide whether a variable or method should be an instance one or a static one? A variable or method that is dependent on a specific instance of the class should be an instance variable or method. A variable or method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius, so the radius is dependent on a specific circle. Therefore, `radius` is an instance variable of the `Circle` class. Since the `getArea` method is dependent on a specific circle, it is an instance method. None of the methods in the `Math` class, such as `random`, `pow`, `sin`, and `cos`, is dependent on a specific instance. Therefore, these methods are static methods. The `main` method is static and can be invoked directly from a class.

**Caution**

It is a common design error to define an instance method that should have been defined as static. For example, the method `factorial(int n)` should be defined as static, as shown next, because it is independent of any specific instance.

```java
public class Test {
    public int factorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++)
            result *= i;
        return result;
    }
}

(a) Wrong design

```java
public class Test {
    public static int factorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++)
            result *= i;
        return result;
    }
}

(b) Correct design
```
Visibility Modifiers

• Visibility modifiers can be used to specify the visibility of a class and its members.
• You can use the public visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as package-private or package-access.
• In addition to the public and default visibility modifiers, Java provides the private and protected modifiers for class members. This section introduces the private modifier. The protected modifier will be introduced in Section 11.13, The protected Data and Methods. The private modifier makes methods and data fields accessible only from within its own class.
package p1;

public class C1 {
    public int x;
    int y;
    private int z;

    public void m1() {
    }
    void m2() {
    }
    private void m3() {
    }
}

package p1;

public class C2 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        can access o.y;
        cannot access o.z;

        can invoke o.m1();
        can invoke o.m2();
        cannot invoke o.m3();
    }
}

package p2;

public class C3 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        cannot access o.y;
        cannot access o.z;

        can invoke o.m1();
        cannot invoke o.m2();
        cannot invoke o.m3();
    }
}

package p1;

class C1 {
    ...
}

package p1;

public class C2 {
    can access C1
}

package p2;

public class C3 {
    cannot access C1;
    can access C2;
}
A visibility modifier specifies how data fields and methods in a class can be accessed from outside the class. There is no restriction on accessing data fields and methods from inside the class.

```java
public class C {
    private boolean x;

    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.x);
        System.out.println(c.convert());
    }

    private int convert() {
        return x ? 1 : -1;
    }
}
```

(a) This is okay because object `c` is used inside the class `C`.

```java
public class Test {
    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.x);
        System.out.println(c.convert());
    }
}
```

(b) This is wrong because `x` and `convert` are private in class `C`.
Data Field Encapsulation

• To prevent direct modifications of data fields, you should declare the data fields private, using the private modifier. This is known as data field encapsulation.

• A private data field cannot be accessed by an object from outside the class that defines the private field. However, a client often needs to retrieve and modify a data field. To make a private data field accessible, provide a get method to return its value. To enable a private data field to be updated, provide a set method to set a new value.
Note
Colloquially, a get method is referred to as a getter (or accessor), and a set method is referred to as a setter (or mutator).

A get method has the following signature:

```
public returnType getPropertyName()
```

If the returnType is boolean, the get method should be defined as follows by convention:

```
public boolean isPropertyName()
```

A set method has the following signature:

```
public void setPropertyName(dataType propertyValue)
```
Passing Objects to Methods

- Passing an object to a method is to pass the reference of the object.

You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. The following code passes the `myCircle` object as an argument to the `printCircle` method:

```java
1 public class Test {
2   public static void main(String[] args) {
3     // CircleWithPrivateDataFields is defined in Listing 8.9
4     CircleWithPrivateDataFields myCircle = new CircleWithPrivateDataFields(5.0);
5     printCircle(myCircle);
6   }
7 }
8
9 public static void printCircle(CircleWithPrivateDataFields c) {
10   System.out.println("The area of the circle of radius " + c.getRadius() + " is " + c.getArea());
11 }
12}
```

Java uses exactly one mode of passing arguments: pass-by-value. In the preceding code, the value of `myCircle` is passed to the `printCircle` method. This value is a reference to a `Circle` object.
Array of Objects

Chapter 6, Single-Dimensional Arrays, described how to create arrays of primitive type elements. You can also create arrays of objects. For example, the following statement declares and creates an array of ten Circle objects:

```java
Circle[] circleArray = new Circle[10];
```

To initialize `circleArray`, you can use a `for` loop like this one:

```java
for (int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle();
}
```

An array of objects is actually an array of reference variables. So, invoking `circleArray[1].getArea()` involves two levels of referencing, as shown in Figure 8.19. `circleArray` references the entire array; `circleArray[1]` references a Circle object.

**Note**

When an array of objects is created using the `new` operator, each element in the array is a reference variable with a default value of `null`. 

![Diagram of object array](image)
Shadowing a `class` variable

- We can reference a `class` variable defined inside the `same class` without using the `class` name. Example:

```java
public class ClassVar2 {
  public static double a;  // <----- Class variable

  public static void main(String[] args) {
    // Body of method "main"
    a = 3.1415;  // We can omit the classname in this method
    System.out.println(a);
  }
}
```

- **Shadowing a class variable:**

  - When the method has a local variable or a parameter variable defined in a scope whose `name` is `equal` to the name of the class variable, then:

    - The `class` variable with the same `name` can no longer `accessible` with the `short hand notation` in that `scope` !!!

- Example:

```java
public class ClassVar4 {
  public static double a = 3.1415;

  public static void main(String[] args) {
    boolean a = true;  // Class var a now "shadowed"
    System.out.println(a);  // prints `true`
    System.out.println(ClassVar4.a);  // prints `3.1415`
  }
}
```
Shadowing *instance* variables

- Recall that we can omit the *implicit parameter* `this` to access an instance variable.

  (This is the *short hand notation* for instance variables)

- Shadowing can also occur with *instance variable* when a *parameter variable* or a *local variable* is defined inside an instance method that has the *same name* as an instance variable.

Example:

```java
public class BankAccount {
    public int accNum; // Shadowed!
    public String name; // Shadowed!
    public double balance; // Shadowed!

    /* ________________________
       convToString(): return a String containing
       information of BankAccount *
    */
    public String convToString() {
        return "Account number: " + accNum + 
                "; Name: " + name + 
                "; Balance: " + balance;
    }

    /* ______________________
       deposit(balance): Add "parameter balance" to balance *
    */
    public void deposit(double balance) {
        balance += balance; // Add parameter balance to (this.)balance
    }

    /* ______________________
       withdraw(balance): Subtract "parameter balance" from balance *
    */
    public void withdraw(double balance) {
        if (balance >= balance)
            balance -= balance; // Subtract parameter balance from (this.)balance
    }
}...
```
Problem:

- The same variable name balance is used to refer to 2 different variables:
  - The parameter variable balance
  - The instance variable (this.)balance

- The Java variable referencing rule (see: click here) will associate the variable name balance with the parameter variable balance.

Note:

- The Java compiler will not report any error because the statement is syntactically correct
Java's automatic conversion rule for `number ⇒ string`:

When the `+` operator is used between:
- a `number` and
- a `string`

Then:
- the `number` is automatically converted to a `string`
- The `+` operator is then applied on 2 strings (i.e., the `+` operator is a concatenation!)
Automatic conversion of variables of a *user-defined* type to String

- Automatic conversion of variables of a *user-defined* type to string
  - Java also provides an *automatic conversion feature* when the `+` operator is used between:
    - *a user-defined type variable* (= object) and *a string typed data*

  **Java's automatic conversion rule for object → string:**

  - When the `+` operator is used between:
    - *a object reference variable and*
    - *a string*

  Then:
    - *the object referred to by the object reference variable is automatically converted to a string*
    - *The + operator is then applied to 2 strings* (i.e., the + operator will be a string concatenation!)

- Example:

```java
public class Class10 {
    public static void main(String[] args) {
        String x;
        BankAccount stud = new BankAccount(123, "John", 1000);
        x = "Test " + stud;  // First converts stud to a String
        System.out.println(x);
    }
}
```

**Output:**

```
Test BankAccount$130019b
```

The string *BankAccount$130019b* is the *result of the conversion of the object stud into the String type*
The `String` class

- `String` is a *user-defined class* that is created by the *programmers* who implemented the *Java library*.
- Here is the *beginning* of the description of the `string` class in the *Java documentation*:

```java
java.lang

Class String

    java.lang.Object
    └ java.lang.String

All Implemented Interfaces:
    Serializable, CharSequence, Comparable<String>

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence

The `String` class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.
- **Defining** `String` variables:

- Using the **object definition syntax** (See: [click here](#))

```java
String s; // s is an identifier
s = new String("Hello World "); // creates a String object and calls a constructor method
```

- The **designer** of the Java language wanted to treat `Strings` special and they made the Java **compiler** recognize a **short hand notation** for string creation:

```java
String s; // s is an identifier
s = "Hello World "; // Short hand for: new String("Hello World");
```

```
public static void main(String[] args)
{
    ...
    String s;
    s = "Hello";
    ...
}
```

---

**Recall that:**

- `s = "Hello";` is short hand for:
  - `s = new String("Hello");`
• Recursion: a special divide and conquer problem solving technique

  ◦ Because there a smaller problem of the same kind embedded inside a recursion problem, there is special divide and conquer solution procedure for recursive problem.

  I like to describe this recursive solution procedure as:

  • the "lazy man's method to solve a problem...."

  ◦ An un-scientific (hopefully more understandable) way to describe the recursive solution method:

  • If you want to solve a problem of size n that has the recursive property, then:

    • Hire (or delegate) someone else to solve the problem of size n-1, size n-2, etc...

    (You would then wait for that person to come back and give you the solution)

    • When you receive the solution for the problem of size n-1, size n-2, etc..., you will then use these solutions to solve the original problem.
The general form of the recursive problem solving algorithm

- The recursive algorithm technique can be expressed in general as follows:

```
ReturnType solveProblem( n )
{
    if ( n is one of the base cases )
    {
        return ( the readily available solution for that base case );
    }
    else
    {
        /* ---------------------------------------------------------------
         * Delegate the problem:
         *     hire someone to solve a (one or more) smaller problems
         * --------------------------------------------------------------- *
         *  sol1 = solveProblem ( n-1 );  // n-1 < n
         *  sol2 = solveProblem ( n-2 );  // n-2 < n
         *  ...
         * ---------------------------------------------------------------
         * Use the solutions of the smaller problems n-1, n-2, ...
         * to solve my problem:
         * --------------------------------------------------------------- *
         *  mySol = Solve "problem(n)" using sol1, sol2, ...
         *--------------------------------------------------------------- *
        return ( mySol );
    }
}
```

- **Where** can you use the recursive algorithm technique?

- The recursive algorithm technique can be used to solve problems that have the recursive property.

- **Notes:**

  - I can only give you the general form of the recursive algorithm technique... because
  - Each problem with the recursive property is solved differently...
• **Recursive methods**

- Consider the above **algorithm**:

```
ReturnType solveProblem( n )
{
    if ( n is one of the base cases )
    {
        return ( the readily available solution for that base case );
    }
    else
    {
        /* -----------------------------------------------
        Delegate the problem:
        hire someone to solve a (one or more) smaller problems
        ----------------------------------------------- */
        sol1 = solveProblem ( n-1 ); // n-1 < n
        sol2 = solveProblem ( n-2 ); // n-2 < n
        ...;
        /* -----------------------------------------------
        Use the solutions of the smaller problems n-1, n-2, ...
        to solve my problem:
        ----------------------------------------------- */
        mySol = Solve “problem(n)” using sol1, sol2, ...
        return ( mySol );
    }
}
```

**Notable fact:**

- When you **implement** the algorithm as a **Java method**, the resulting method will **invoke itself**!!!
- We call these kind of **methods**: **recursive methods**

**Definition:**

- **Recursive method** = a **method** that **invokes itself**

We will see some **recursive methods** in the next few webpages.
Putting a class into a package

- Commonly used organization structure
  - Organization structure
    - The hierarchical structure is a commonly used structure for organization
  - Examples of organizing topics into hierarchies:

![Diagram showing hierarchies of classes and geography](image)
Java programming rule:

- A `class ClassA` that is stored inside a `package packageX` must be stored inside a (UNIX) directory named `PackageX`.

Illustrated:

```
package packageX;
public class ClassA {
    ***
}
```

```
package packageX;
public class ...
{
    ***
}
```
Example:

- The **Hello World Java program** that is defined inside the **package myLib** must be stored inside a directory named **myLib**.
• Compiling and running methods in a class stored inside a package
  ○ When compiling a class or running (executing) methods of a class inside a package, your working directory should be:

    - the directory that contains the directory of the package.

Example:

```
package packageX

(ClassA.java)
package packageX;
public class ClassA
{
    ...
}

package packageX;
public class ...
{
    ...
}
```

- To compile a java program file in package packageX, use the command:

  ```
cd Dir  
java packageX/javaProgram.java
  ```

- To run a java program file in package packageX, use the command:

  ```
cd Dir  
java packageX.javaProgram
  ```