Lecture 24
Reference Data Fields and the null Value

The data fields can be of reference types. For example, the following `Student` class contains a data field `name` of the `String` type. `String` is a predefined Java class.

```java
class Student {
    String name; // name has the default value null
    int age; // age has the default value 0
    boolean isScienceMajor; // isScienceMajor has default value false
    char gender; // gender has default value '\u0000'
}
```

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:
class Test {
    public static void main(String[] args) {
        System.out.println("name? " + student.name);
        System.out.println("age? " + );
        System.out.println("isScienceMajor? " + );
        System.out.println("gender? " + );
    }
}

The following code has a compile error, because the local variables x and y are not initialized:

class Test {
    public static void main(String[] args) {
        int x; // x has no default value
        String y; // y has no default value
        System.out.println("x is " + );
        System.out.println("y is " + );
    }
}

Caution
NullPointerException is a common runtime error. It occurs when you invoke a method on a reference variable with a null value. Make sure you assign an object reference to the variable before invoking the method through the reference variable.
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

![Figure 8.7](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.

![Figure 8.8](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.

**Object type assignment $c1 = c2$**

**Before:**
- $c1$
- $c2$

**After:**
- $c1$
- $c2$

**Figure 8.9** Reference variable $c2$ is copied to variable $c1$. 
Note
As illustrated in Figure 8.9, after the assignment statement `c1 = c2`, `c1` points to the same object referenced by `c2`. The object previously referenced by `c1` is no longer useful and therefore is now known as *garbage*. Garbage occupies memory space, so the Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

Tip
If you know that an object is no longer needed, you can explicitly assign `null` to a reference variable for the object. The JVM will automatically collect the space if the object is not referenced by any reference variable.
Using Classes from the Java Library

• The Date Class

```
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()
+Random(seed: long)
+nextInt(): int
+nextInt(n: int): int
+nextLong(): long
+nextDouble(): double
+nextFloat(): float
+nextBoolean(): boolean

Constructs a Random object with the current time as its seed.
Constructs a Random object with a specified seed.
Returns a random int value.
Returns a random int value between 0 and n (excluding n).
Returns a random long value.
Returns a random double value between 0.0 and 1.0 (excluding 1.0).
Returns a random float value between 0.0F and 1.0F (excluding 1.0F).
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.
```java
static int numberOfObjects;

static int getNumberOfObjects() {
    return numberOfObjects;
}
```

**UML Notation:**
- Underline: static variables or methods

**Diagram:**
- `circle1`: `radius = 1`, `numberOfObjects = 2`
- `circle2`: `radius = 5`, `numberOfObjects = 2`

**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.

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

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

```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;
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.x);
        System.out.println(c.convert());
    }
}
```
Caution
The **private** modifier applies only to the members of a class. The **public** modifier can apply to a class or members of a class. Using the modifiers **public** and **private** on local variables would cause a compile error.

Note
In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a **private constructor**. For example, there is no reason to create an instance from the **Math** class, because all of its data fields and methods are static. To prevent the user from creating objects from the **Math** class, the constructor in **java.lang.Math** is defined as follows:

```java
private Math() {
}
```
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:

```java
public returnType getPropertyName()
```

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

```java
public boolean isPropertyName()
```

A **set** method has the following signature:

```java
public void setPropertyName(dataType propertyValue)
```
toString function

• Print the object as a string.
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 }
13
```

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 showing the reference structure of an array of Circle objects]

- `circleArray`: Reference to the entire array of Circle objects.
- `circleArray[0]`: Reference to the first Circle object.
- `circleArray[1]`: Reference to the second Circle object.
- `circleArray[9]`: Reference to the last Circle object.
- Each Circle object is represented as a separate node in the diagram.