Lecture 23
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;
    }
}
```
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);
```
A class normally provides a constructor without arguments (e.g., \texttt{Circle()}). Such a constructor is referred to as a \textit{no-arg} or \textit{no-argument constructor}.

A class may be defined without constructors. In this case, a public no-arg constructor with an empty body is implicitly defined in the class. This constructor, called a \textit{default constructor}, is provided automatically \textit{only if no constructors are explicitly defined in the class}.

8.5 What are the differences between constructors and methods?  
8.6 When will a class have a default constructor?
• Copy constructor.
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.
Note
An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. Therefore, it is fine, for simplicity, to say that myCircle is a Circle object rather than use the longer-winded description that myCircle is a variable that contains a reference to a Circle object.

Note
Arrays are treated as objects in Java. Arrays are created using the new operator. An array variable is actually a variable that contains a reference to an array.
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.

For example, `myCircle.radius` references the radius in `myCircle`, and `myCircle.getArea()` invokes the `getArea` method on `myCircle`. Methods are invoked as operations on objects.

The data field `radius` is referred to as an instance variable, because it is dependent on a specific instance. For the same reason, the method `getArea` is referred to as an instance method, because you can invoke it only on a specific instance. The object on which an instance method is invoked is called a calling object.
**Caution**
Recall that you use `Math.methodName(arguments)` (e.g., `Math.pow(3, 2.5)`) to invoke a method in the `Math` class. Can you invoke `getArea()` using `Circle.getArea()`? The answer is no. All the methods in the `Math` class are static methods, which are defined using the `static` keyword. However, `getArea()` is an instance method, and thus nonstatic. It must be invoked from an object using `objectRefVar.methodName(arguments)` (e.g., `myCircle.getArea()`). Further explanation is given in Section 8.7, Static Variables. Constants, and Methods.

**Note**
Usually you create an object and assign it to a variable, and then later you can use the variable to reference the object. Occasionally an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable using the syntax:

```java
new Circle();
```

or

```java
System.out.println("Area is " + new Circle().getArea());
```

The former statement creates a `Circle` object. The latter creates a `Circle` object and invokes its `getArea` method to return its area. An object created in this way is known as an anonymous object.
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:
**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 $c_1 = c_2$ copies the reference of $c_2$ into $c_1$ for reference variables. After the assignment, variables $c_1$ and $c_2$ refer to the same object.

![Diagram](image_url)

**Figure 8.9** Reference variable $c_2$ is copied to variable $c_1$. 
**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
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.util.Random
+Random()
+Random(seed: long)
+nextInt(): int
+nextInt(n: int): int
+nextLong(): long
+nextDouble(): double
+nextFloat(): float
+nextBoolean(): boolean
```

- `Random()`: Constructs a `Random` object with the current time as its seed.
- `Random(seed: long)`: Constructs a `Random` object with a specified seed.
- `nextInt()`: Returns a random `int` value.
- `nextInt(n: int)`: Returns a random `int` value between 0 and n (excluding n).
- `nextLong()`: Returns a random `long` value.
- `nextDouble()`: Returns a random `double` value between 0.0 and 1.0 (excluding 1.0).
- `nextFloat()`: Returns a random `float` value between 0.0F and 1.0F (excluding 1.0F).
- `nextBoolean()`: Returns a random `boolean` value.