

Chapter 9 Objects and Classes



Objectives

- ❑ To describe objects and classes, and to use classes to model objects (§9.2).
- ❑ To use UML graphical notations to describe classes and objects (§9.2).
- ❑ To demonstrate defining classes and creating objects (§9.3).
- ❑ To create objects using constructors (§9.4).
- ❑ To access data fields and invoke functions using the object member access operator (.) (§9.5).
- ❑ To separate a class definition from a class implementation (§9.6).
- ❑ To prevent multiple inclusions of header files using the **#ifndef** inclusion guard directive (§9.7).
- ❑ To know what inline functions in a class are (§9.8).
- ❑ To declare private data fields with appropriate **get** and **set** functions for data field encapsulation and make classes easy to maintain (§9.9).
- ❑ To understand the scope of data fields (§9.10).
- ❑ To apply class abstraction to develop software (§9.11).

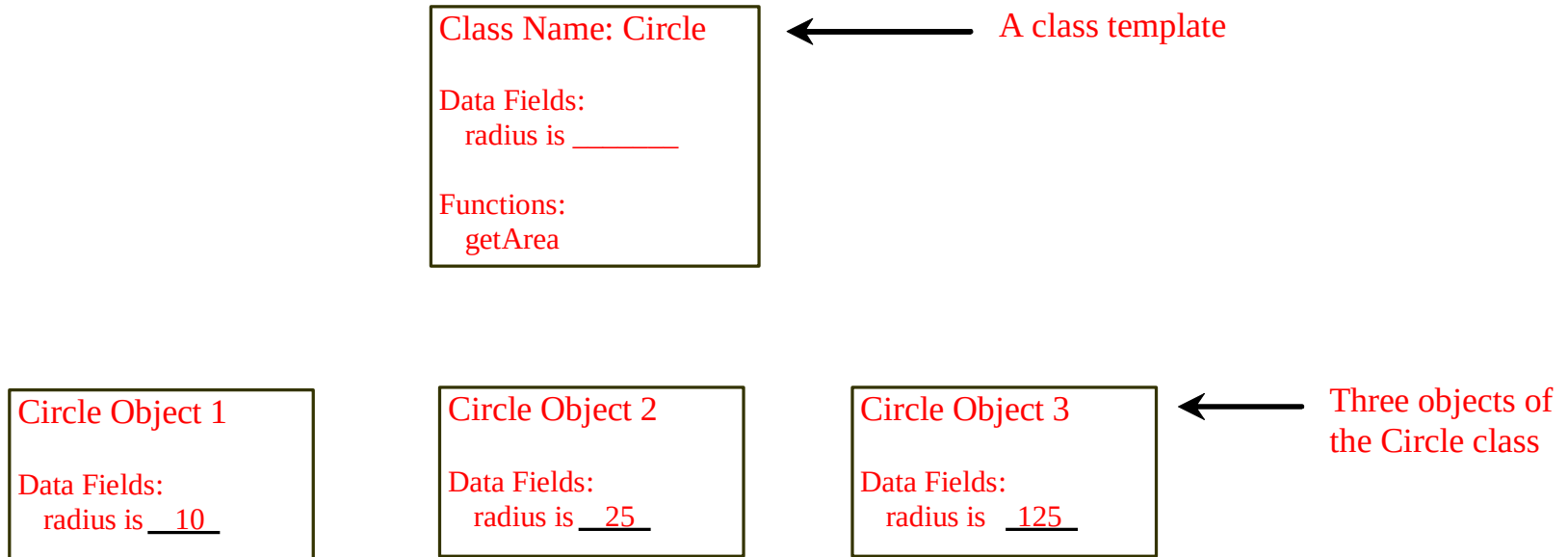


OO Programming Concepts

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behaviors. The *state* of an object consists of a set of *data fields* (also known as *properties*) with their current values. The *behavior* of an object is defined by a set of functions.



Objects



An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

Classes

Classes are constructs that define objects of the same type. A class uses variables to define data fields and functions to define behaviors.

Additionally, a class provides a special type of functions, known as constructors, which are invoked to construct objects from the class.



Classes

```
class Circle
{
public:
    // The radius of this circle
    double radius;

    // Construct a circle object
    Circle()
    {
        radius = 1;
    }

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

    // Return the area of this circle
    double getArea()
    {
        return radius * radius * 3.14159;
    }
};
```

← Data field

← Constructors

← Function



Classes

C++11: data field
can be initialized in
C++

```
class Circle
{
public:
    // 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 * 3.14159;
    }
};
```

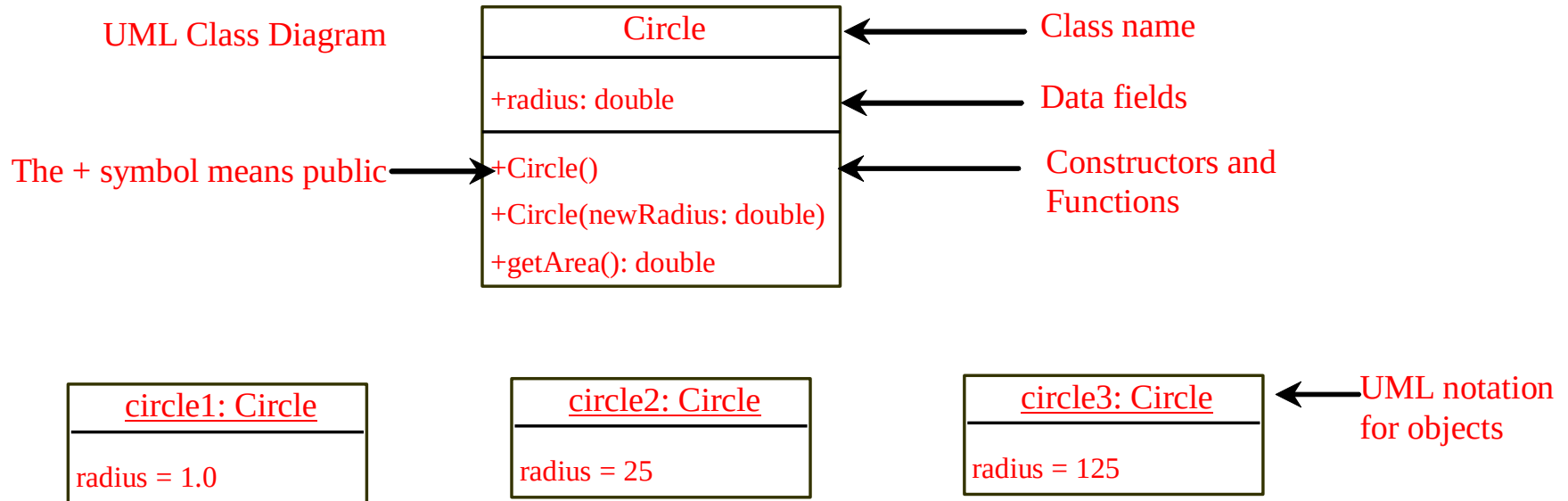
← Data field

← Constructors

← Function

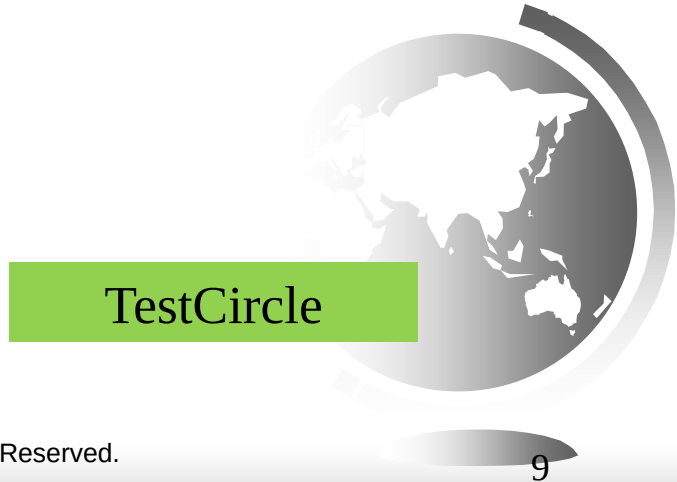


UML Class Diagram



A Simple Circle Class

- Objective: Demonstrate creating objects, accessing data, and using functions.



TestCircle

Example: Defining Classes and Creating Objects

TV	
channel: int volumeLevel: int on: bool	
+TV() +turnOn(): void +turnOff(): void +setChannel(newChannel: int): void +setVolume(newVolumeLevel: int): void +channelUp(): void +channelDown(): void +volumeUp(): void +volumeDown(): void	

The current channel (1 to 120) of this TV.

The current volume level (1 to 7) of this TV.

Indicates whether this TV is on/off.

Constructs a default TV object.

Turns on this TV.

Turns off this TV.

Sets a new channel for this TV.

Sets a new volume level for this TV.

Increases the channel number by 1.

Decreases the channel number by 1.

Increases the volume level by 1.

Decreases the volume level by 1.



TV

Constructors

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

A class normally provides a constructor without arguments (e.g., Circle()). Such constructor is called a *no-arg* or *no-argument constructor*.

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



Constructors, cont.

A constructor with no parameters is referred to as a *no-arg constructor*.

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors play the role of initializing objects.



Object Names

In C++, you can assign a name when creating an object. A constructor is invoked when an object is created. The syntax to create an object using the no-arg constructor is

```
ClassName objectName;
```

For example,

```
Circle circle1;
```



Constructing with Arguments

The syntax to declare an object using a constructor with arguments is

```
ClassName objectName(arguments);
```

For example, the following declaration creates an object named circle2 by invoking the Circle class's constructor with a specified radius 5.5.

```
Circle circle2(5.5);
```



Access Operator

After an object is created, its data can be accessed and its functions invoked using the dot operator (.), also known as the *object member access operator*:

objectName.dataField references a data field in the object.

objectName.function(arguments) invokes a function on the object.



Naming Objects and Classes

When you declare a custom class, capitalize the first letter of each word in a class name; for example, the class names Circle, Rectangle, and Desk. The class names in the C++ library are named in lowercase. The objects are named like variables.



Class is a Type

You can use primitive data types to declare variables. You can also use class names to declare object names. In this sense, a class is also a data type.



Memberwise Copy

In C++, you can also use the assignment operator `=` to copy the contents from one object to the other. By default, each data field of one object is copied to its counterpart in the other object. For example,

```
circle2 = circle1;
```

copies the radius in circle1 to circle2. After the copy, circle1 and circle2 are still two different objects, but with the same radius.



Constant Object Name

Object names are like array names. Once an object name is declared, it represents an object. It cannot be reassigned to represent another object. In this sense, an object name is a constant, though the contents of the object may change.



Anonymous Object

Most of the time, you create a named object and later access the members of the object through its name.

Occasionally, you may create an object and use it only once. In this case, you don't have to name the object. Such objects are called *anonymous objects*.

The syntax to create an anonymous object using the no-arg constructor is

```
ClassName()
```

The syntax to create an anonymous object using the constructor with arguments is

```
ClassName(arguments)
```



Class Replaces struct

The C language has the struct type for representing records. For example, you may define a struct type for representing students as shown in (a).

```
struct Student
{
    int id;
    char firstName[30];
    char mi;
    char lastName[30];
};
```

(a)

```
class Student
{
public:
    int id;
    char firstName[30];
    char mi;
    char lastName[30];
};
```

(b)

Separating Definition from Implementation

C++ allows you to separate class definition from implementation. The class definition describes the contract of the class and the class implementation implements the contract. The class declaration simply lists all the data fields, constructor prototypes, and the function prototypes. The class implementation implements the constructors and functions. The class declaration and implementation are in two separate files. Both files should have the same name, but with different extension names. The class declaration file has an extension name .h and the class implementation file has an extension name .cpp.

Circle.h

Circle.cpp

TestCircleWithHeader.cpp

Preventing Multiple Inclusions

It is a common mistake to include the same header file in a program multiple times inadvertently. Suppose Head1.h includes Circle.h and TestHead1.cpp includes both Head1.h and Circle.h, as shown in the next slide.



Preventing Multiple Declarations

Head1.h

```
#include "Circle.h"  
// Other code in Head1.h omitted
```

TestHead1.cpp

```
#include "Circle.h"  
#include "Head1.h"  
int main()  
{  
    // Other code in TestHead.cpp  
    // omitted  
}
```



inclusion guard

```
#ifndef CIRCLE_H
#define CIRCLE_H
class Circle
{
public:
    // The radius of this circle
    double radius;
    // Construct a default circle object
    Circle();
    // Construct a circle object
    Circle(double);
    // Return the area of this circle
    double getArea();
}; // Semicolon required
#endif
```



Inline Declaration

Chapter 6 introduced how to improve function efficiency using inline functions. Inline functions play an important role in class declarations. When a function is implemented inside a class declaration, it automatically becomes an inline function. This is also known as *inline declaration*.



Inline Declaration Example

For example, in the following declaration for class A, the constructor and function f1 are automatically inline functions, but function f2 is a regular function.

```
class A
{
public:
    A()
    {
        // do something;
    }
    double f1()
    {
        // return a number
    }

    double f2();
};
```



Inline Functions in Implementation File

There is another way to declare inline functions for classes. You may declare inline functions in the class's implementation file. For example, to declare function f2 as an inline function, precede the inline keyword in the function header as follows:

```
// Implement function as inline
```

```
inline double A::f2()
```

```
{
```

```
    // return a number
```

```
}
```



Inline Declarations?

Short functions are good candidates for inline functions, but long functions are not.



Data Field Encapsulation

The data fields radius in the Circle class in Listing 9.1 can be modified directly (e.g., circle1.radius = 5). This is not a good practice for two reasons:

- First, data may be tampered.
- Second, it makes the class difficult to maintain and vulnerable to bugs. Suppose you want to modify the Circle class to ensure that the radius is non-negative after other programs have already used the class. You have to change not only the Circle class, but also the programs that use the Circle class. Such programs are often referred to as *clients*. This is because the clients may have modified the radius directly (e.g., myCircle.radius = -5).

Accessor and Mutator

Colloquially, a get function is referred to as a *getter* (or *accessor*), and a set function is referred to as a *setter* (or *mutator*).

A get function has the following signature:

```
returnType getPropertyName()
```

If the returnType is bool, the get function should be defined as follows by convention:

```
bool isPropertyName()
```

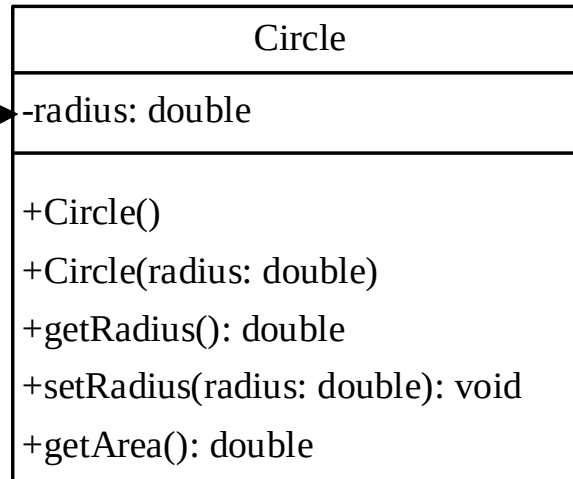
A set function has the following signature:

```
public void setPropertyName(dataType propertyValue)
```



Example: New Circle Class

The - sign indicates
private modifier



The radius of this circle (default: 1.0).

Constructs a default circle object.

Constructs a circle object with the specified radius.

Returns the radius of this circle.

Sets a new radius for this circle.

Returns the area of this circle.

CircleWithPrivateDataFields.h

CircleWithPrivateDataFields.cpp

TestCircleWithPrivateDataFields



The Scope of Variables

Chapter 6, “Functions,” discussed the scope of global variables and local variables. Global variables are declared outside all functions and are accessible to all functions in its scope. The scope of a global variable starts from its declaration and continues to the end of the program. Local variables are defined inside functions. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.



The Scope of Variables

The data fields are declared as variables and are accessible to all constructors and functions in the class. In this sense, data fields are like global variables. However, data fields and functions can be declared in any order in a class. For example, all the following declarations are the same:

```
class Circle
{
public:
    Circle();
    Circle(double);
    double getArea();
    double getRadius();
    void setRadius(double);

private:
    double radius;
};
```

(a)

```
class Circle
{
public:
    Circle();
    Circle(double);

private:
    double radius;

public:
    double getArea();
    double getRadius();
    void setRadius(double);
};
```

(b)

```
class Circle
{
private:
    double radius;

public:
    double getArea();
    double getRadius();
    void setRadius(double);

public:
    Circle();
    Circle(double);
};
```

(c)

The Scope of Variables

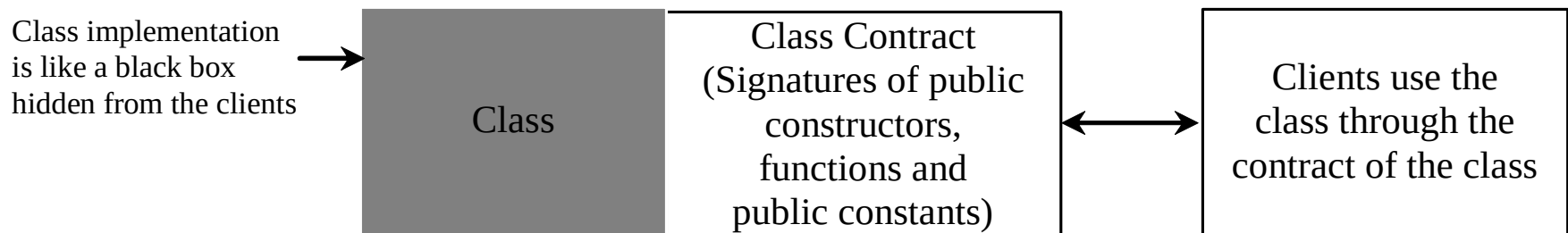
Local variables are declared and used inside a function locally. If a local variable has the same name as a data field, the local variable takes precedence and the data field with the same name is hidden. For example, in the following program in Listing 9.8, x is defined as a data field and as a local variable in the function.

HideDataField



Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



Example: The Loan Class

Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1)
-loanAmount: double	The loan amount (default: 1000).
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate to this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years to this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount to this loan.
+getMonthlyPayment(): double	Returns the monthly payment of this loan.
+getTotalPayment(): double	Returns the total payment of this loan.

Loan.h

Loan.cpp

TestLoanClass

