17. Objects and Classes (Part 1)

Java

Fall 2009

Instructor: Dr. Masoud Yaghini

Outline

- Introduction
- Defining Classes for Objects
- Constructors
- Creating Objects
- Accessing an Object's Data and Methods
- An Example: CreatObjectDemo.java
- An Example: TestCircle1.java
- Default Values of Data Fields
- Differences Between Variables of Primitive Types and Reference Types
- Using Classes from the Java Library
- References

Introduction

Procedural Programming Languages

- Programming in **procedural languages** like C, Pascal, BASIC, and COBOL involves:
  - Choosing data structures,
  - Designing algorithms, and
  - Translating algorithms into code.
- In **procedural programming**, **data** and **operations** on the data are separate, and this methodology requires sending data to methods.

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
  - a loan

OO Programming Concepts

- An object has a unique identity, state, and behaviors.
- **State**
  - The state of an object consists of a set of **data fields** (also known as **properties**) with their current values.
  - The state defines the object.
- **Behavior**
  - The behavior of an object is defined by a set of methods.
  - Invoking a method on an object means that you ask **the object to perform a task**.
  - The behavior defines what the object does.

Defining Classes for Objects

Defining Classes for Objects

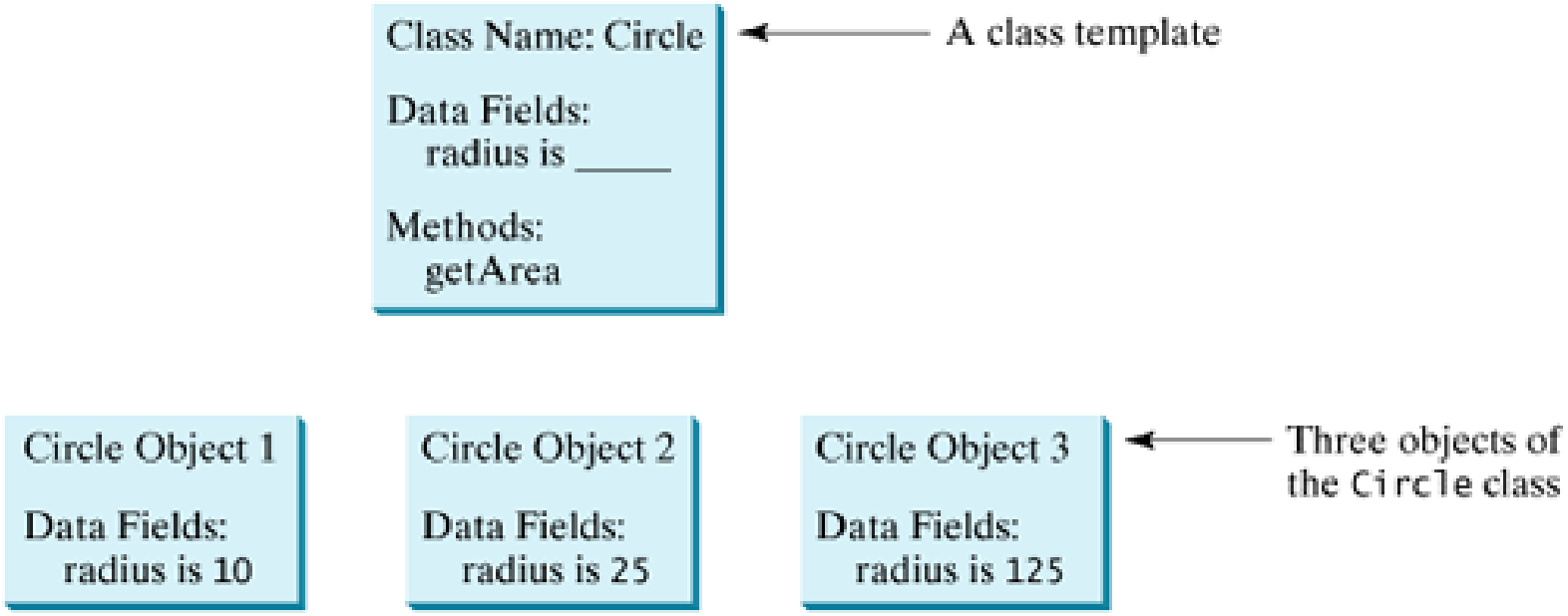
- A circle object, for example, has a data field, radius, which is the property that characterizes a circle.
- One behavior of a circle is that its area can be computed using the method getArea().

Defining Classes for Objects

- **Classes**
  - are templates or blueprints that define objects of the same type
  - A class defines what an object's **data** and **methods** will be.
- An **object** is **an instance** of a class.
- You can create many instances of a class.
- **Instantiation**
  - Creating an instance is referred to as **instantiation**.
- The terms **object** and **instance** are often **interchangeable**.

Defining Classes for Objects

- This Figure shows a class named Circle and its three objects.



Defining Classes for Objects

- A **Java class** uses
  - **variables** to define **data fields** and
  - **methods** to define **behaviors**.
- **Constructors**
  - A class provides methods of a special type, known as **constructors**, which are invoked when a new object is created.
  - A **constructor** is a special kind of method.
  - A **constructor** can perform **any action**, but constructors are designed to perform initializing actions, such as initializing the data fields of objects.

Defining Classes for Objects

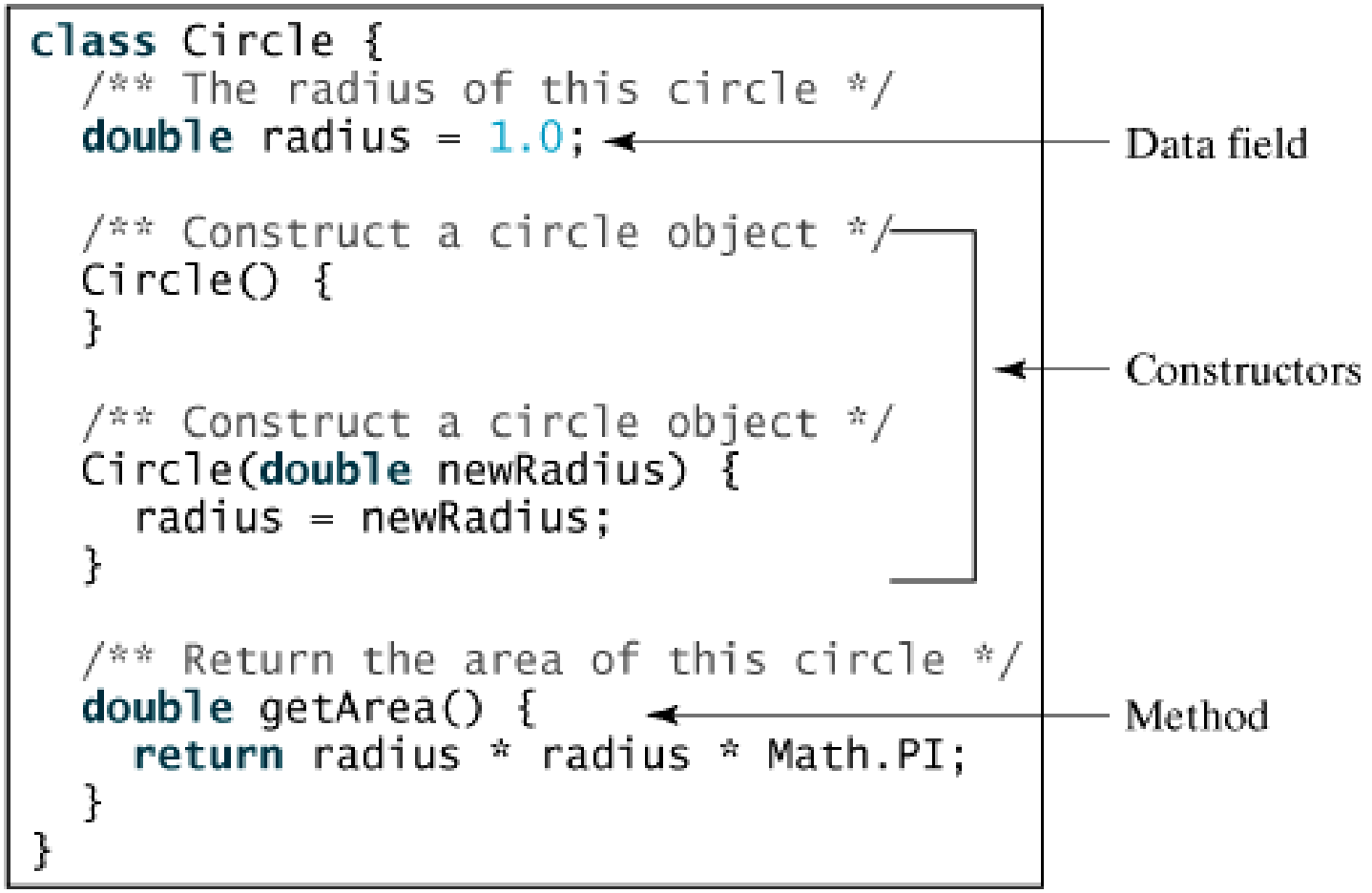
- General form of **class declaration:**

```
class MyClass
{
    // class body: field, constructor, and method declarations
}
```

- The class body (the area between the braces) contains:
  - **declarations for the fields** that provide the state of the class and its objects
  - **constructors** for initializing new objects
  - **methods** to implement the behavior of the class and its objects

Defining Classes for Objects

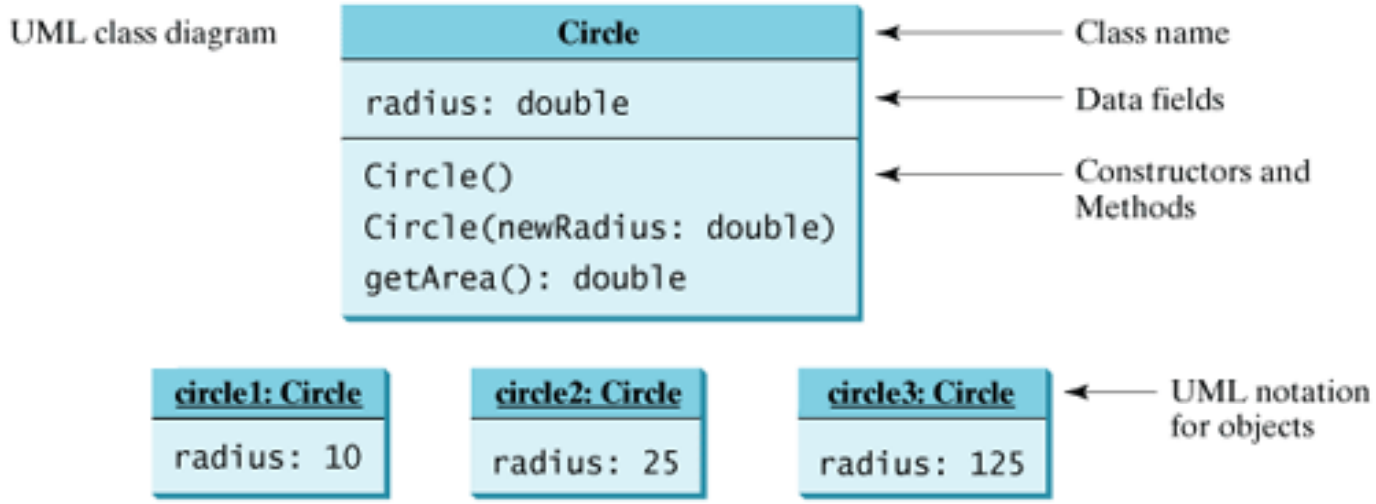
- An example of the Circle class



Defining Classes for Objects

- The Circle class does not have a main method and therefore cannot be run.
- It is merely a definition used to declare and create Circle objects.
- The illustration of class templates and objects in can be standardized using **UML (Unified Modeling Language)** notations.

UML Class Diagram



- The data field is denoted as:
  dataFieldName: dataFieldType
- The constructor is denoted as
  ClassName(parameterName: parameterType)
- The method is denoted as:
  methodName(parameterName: parameterType): returnType

Constructors

Constructors

- **Constructors** are a special kind of methods that are invoked to construct objects.
- The constructor has exactly the **same name** as the defining class.
- Like regular methods, constructors can be overloaded, making it easy to construct objects with different initial data values.

```
Circle()
{
}
Circle(double newRadius)
{
  radius = newRadius;
}
```

Constructors

- To construct an object from a class, invoke a constructor of the class using the new operator, as follows:

  new ClassName(arguments);

- For example:
  - new Circle() creates an object of the Circle class using the first constructor defined in the Circle class
  - new Circle(5) creates an object using the second constructor defined in the Circle class.

Default Constructor

- A constructor with no parameters is referred to as a **no-arg constructor** (e.g., Circle()).
- 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

- **Constructors** are a special kind of method, with three differences:
  - Constructors 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.

Creating Objects

Creating Objects

- To create an object you should:
  - **Declare** an **object reference variable**
    - Any variable of the **class type** can reference to an instance of the class.
  - **Create** an object
  - **Assign** the **object reference** to the **reference variable**

Creating Objects

- To declare an **object reference variable**, use the syntax:

  ClassName objectRefVar;

- Example:

  Circle myCircle;

Creating Objects

- The variable myCircle can reference a Circle object.
- This statement creates an object and assigns its reference to myCircle.

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

Creating Objects

- You can write one statement that combines
  - the **declaration** of an object reference variable,
  - the **creation** of an object, and
  - the **assigning** of the **object reference** to the variable.

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

- An example:

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

Creating Objects

- myCircle is not an object but it is a variable that contains a reference to a Circle object.
- For simplicity, we say that myCircle is a Circle object

Accessing an Object's Data and Methods

Accessing an Object's Data and Methods

- After an object is created, its data can be accessed and its methods invoked using the **dot operator (.)**, also known as the **object member access operator**.
- To access a data field in the object:
  - objectRefVar.dataField
  - e.g., `myCircle.radius`
- To invoke a method on the object:
  - objectRefVar.method(arguments)
  - e.g., `myCircle.getArea()`

Accessing an Object's Data and Methods

- **Instance variable**
  - The data field radius is referred to as an **instance variable** because it is dependent on a specific instance.
- **Instance method**
  - The method getArea is referred to as an **instance method**, because you can only invoke it on a specific instance.
- **Calling object**
  - The object on which an instance method is invoked is referred to as a **calling object**.

Anonymous Object

- You can create an object without explicitly **assigning** it to a variable, as shown below:

```
System.out.println("Area is " + new Circle(5).getArea());
```

- This statement 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**.

An Example: CreatObjectDemo.java

An example

- An example:
  - Point.java
  - Rectangle.java
  - CreateObjectDemo.java
- Here's the output:

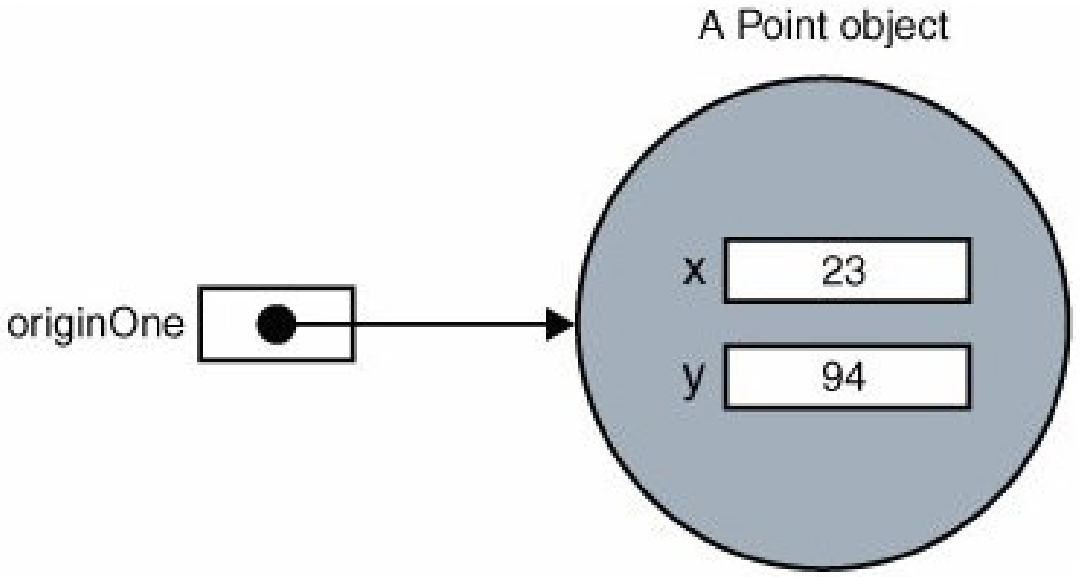
```
Width of rectOne: 100
Height of rectOne: 200
Area of rectOne: 20000
X Position of rectTwo: 23
Y Position of rectTwo: 94
X Position of rectTwo: 40
Y Position of rectTwo: 72
```

An example

- The following statement provides 23 and 94 as values for Point class arguments:

  Point originOne = new Point(23, 94);

- originOne now points to a Point object.



An example

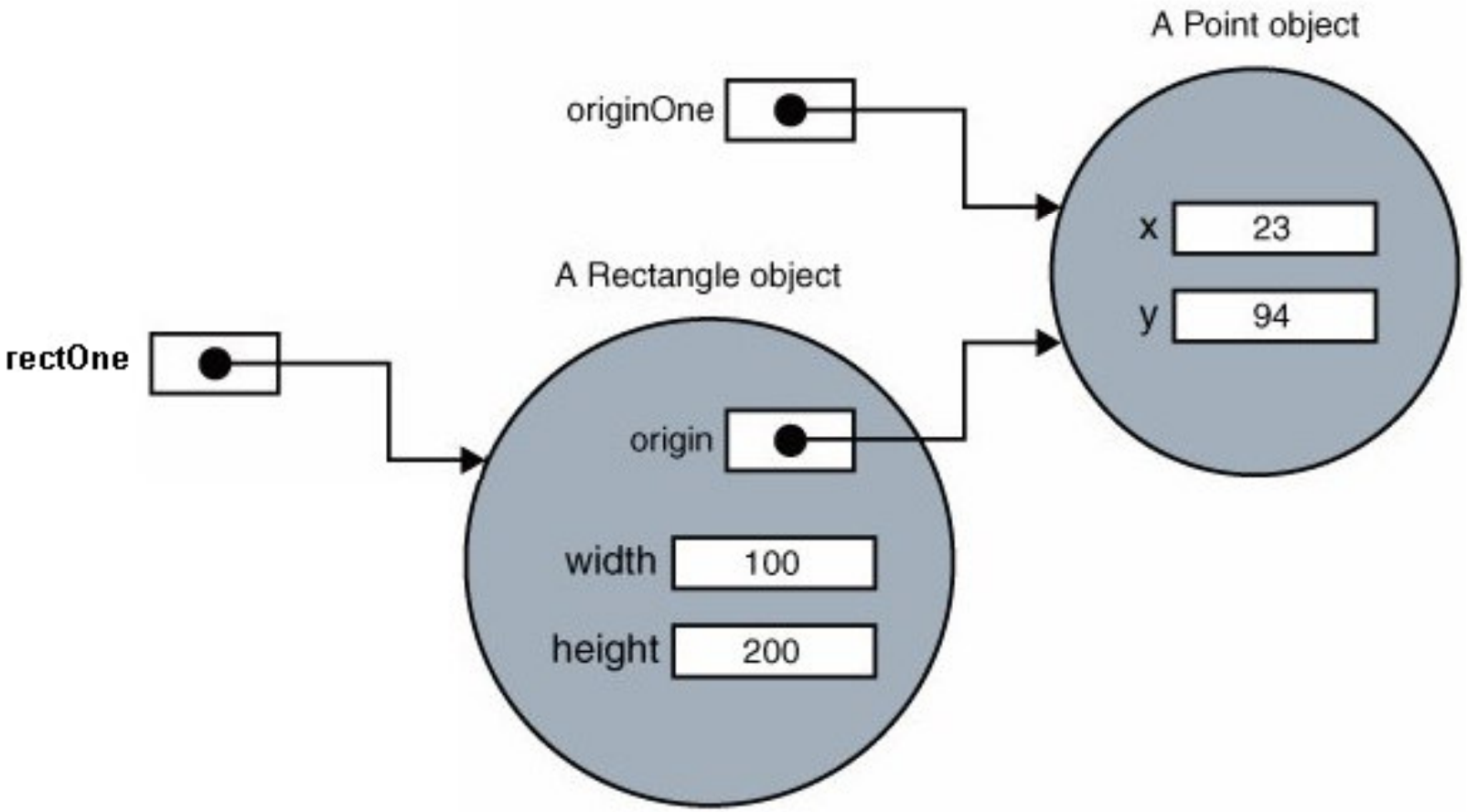
- Rectangle class has different constructors but when the Java compiler encounters the following code:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

- It knows to invoke the constructor in the Rectangle class that requires a Point argument followed by two integer arguments.
- Now there are two references to the same Point object
- An object can have multiple references to it

An example

- rectOne now points to a Rectangle object there are two references to the same Point object:



An example

- The following line of code invokes the Rectangle constructor that requires two integer arguments, which provide the initial values for width and height. And it creates a new Point object whose x and y values are initialized to 0:

  Rectangle rectTwo = new Rectangle(50, 100);

- The Rectangle constructor used in the following statement doesn't take any arguments, so it's called a no-argument constructor:

  Rectangle rect = new Rectangle();

An Example: TestCircle1.java

An example

- An example:
  - TestCircle1.java
- The program constructs a circle object with radius 5 and an object with radius 1 and displays the radius and area of each of the two circles.
- Change the radius of the second object to 100 and display its new radius and area

An example

- The program contains two classes.
- The first class, TestCircle1, is the main class. Its purpose is to test the second class, Circle1.
- Every time you run the program, the JVM invokes the main method in the main class.
- You can put the two classes into one file, but **only one class** in the file can be a public class.
- Furthermore, the public class must have the same name as the file name and the main method must be in a **public class**.

An example

- To write the getArea method in a **procedural programming language** like Pascal, you would pass radius as an argument to the method.
- But in **object-oriented programming**, radius and getArea are defined in the **object**.
- The radius is a data member in the object, which is accessible by the getArea method.
- In **procedural programming languages**, data and methods are **separated**, but in an o**bject-oriented programming language**, data and methods are grouped **together**.

Other way to write the program

- There are many ways to write Java programs.
- For instance, you can combine the two classes in the example into one.
- This demonstrates that you can test a class by simply adding a main method in the same class.
- Example:
  - Circle1.java

An example

- 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 Circle1.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 **non-static**.
- It must be invoked from an object using objectRefVar.methodName(arguments) (e.g., myCircle.getArea()). "

Default Values of Data Fields

Default Values of Data Fields

- The data fields can be of **reference types**.
- For example, the following Student class contains a data field name of the String type.

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

- name is a **reference variable**.
- String is a predefined Java class.

Default Values of Data Fields

- If a data field of a **reference type** does not reference any object, the data field holds a special Java value, **null**.
- The default value of a data field is:
  - null for a **reference type**
  - 0 for a **numeric type**
  - false for a **boolean type**
  - '\u0000' for a **char type**

Default Values of Data Fields

- An example:
  - TestDefaultValue1.java

- The program output:

```
name? null
age? 0
isScienceMajor? false
gender?
```

Default Values of Data Fields

- Java assigns no default value to a local variable inside a method.
- The following program has a **compilation error** because local variables x and y are not initialized.
  - TestDefaultValue2.java

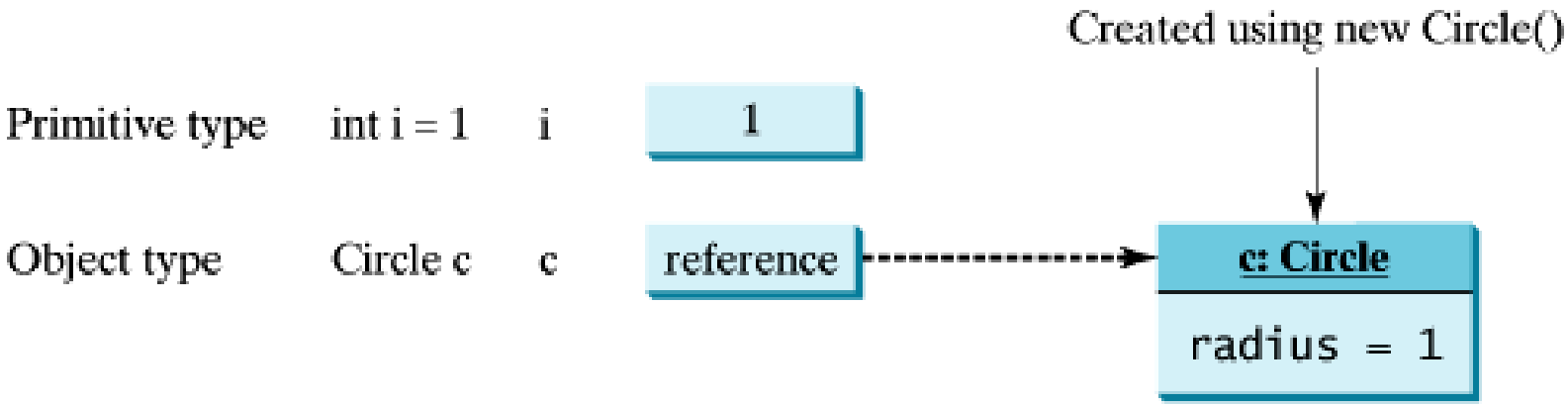
Differences Between Variables of Primitive Types and Reference Types

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.

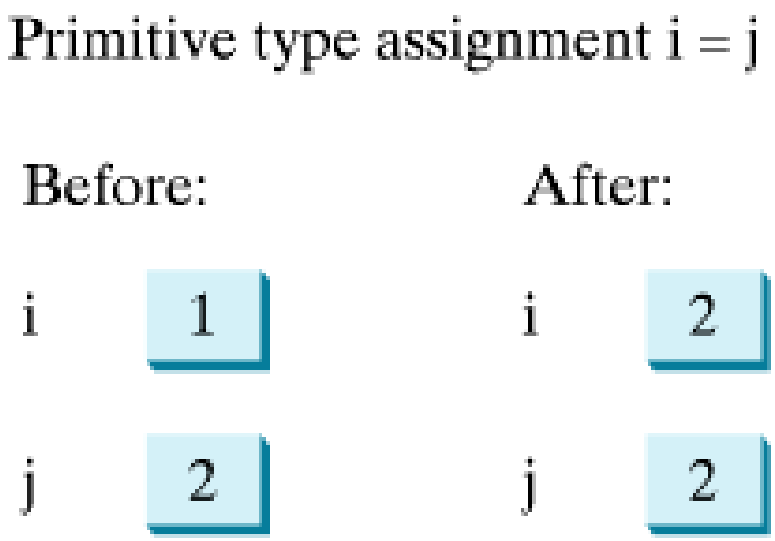
Differences Between Variables of Primitive Types and Reference Types

- 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 the memory.



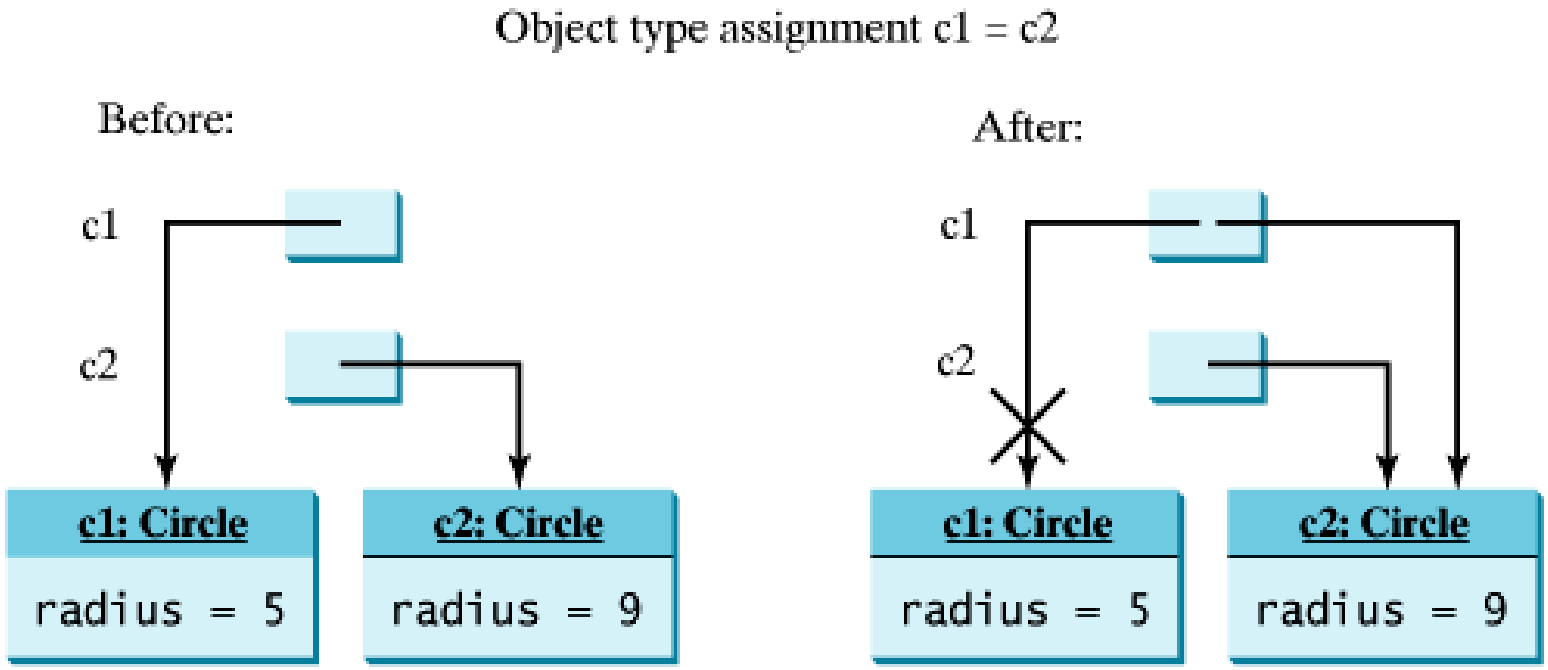
Differences Between Variables of Primitive Types and Reference Types

- 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.



Differences Between Variables of Primitive Types and Reference Types

- For a variable of a reference type, the reference of one variable is assigned to the other variable.



Differences Between Variables of Primitive Types and Reference Types

- 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.
- The JVM detects garbage and automatically reclaims the space it occupies.
- This process is called **garbage collection**.

Differences Between Variables of Primitive Types and Reference Types

- 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 variable.

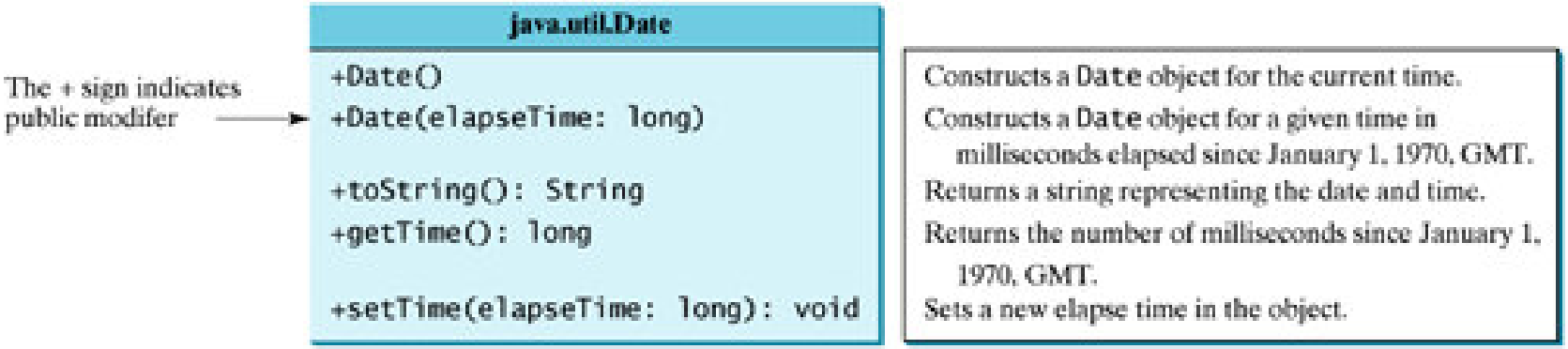
Using Classes from the Java Library

Using Classes from the Java Library

- You will frequently use the classes in the Java library to develop programs.
- This section gives some examples of the classes in the Java library.

The Date Class

- Java provides a system-independent encapsulation of date and time in the java.util.Date class.
- You can use the Date class to create an instance for the current date and time and use its toString method to return the date and time as a string.



The Date Class

- For example, the following code

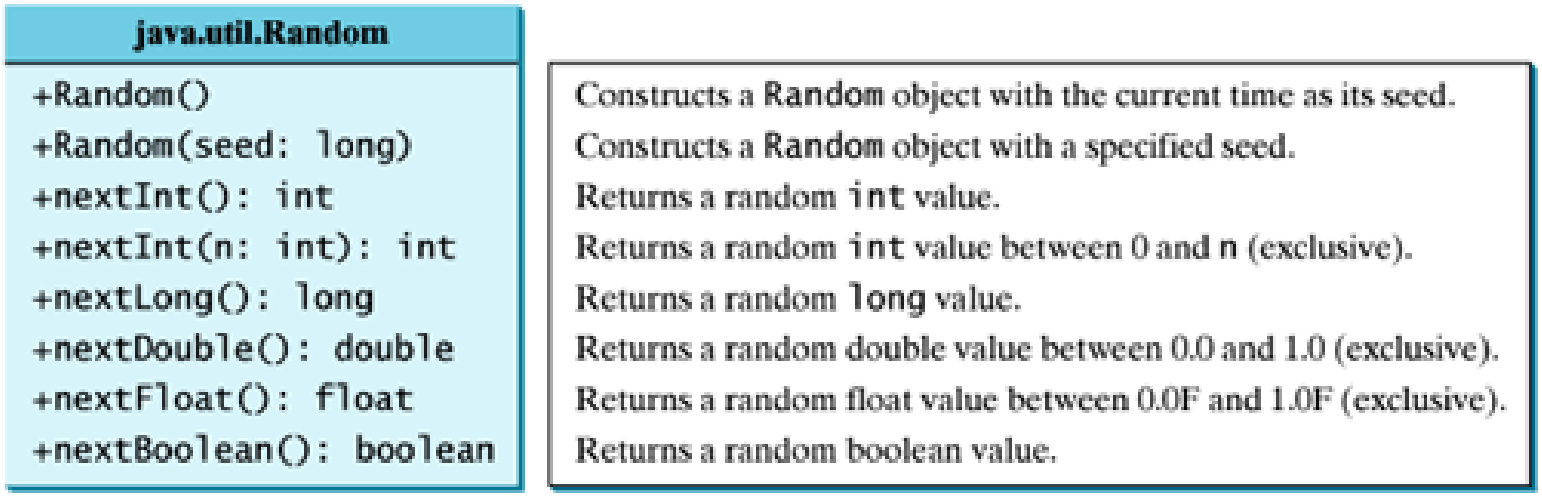
```
java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " +
      date.getTime() + " milliseconds");
System.out.println(date.toString());
```

- displays the output like this:

```
The elapse time since Jan 1, 1970 is
1100547210284 milliseconds
Mon Nov 15 14:33:30 EST 2004
```

The Random Class

- You have used Math.random() to obtain a random double value between 0.0 and 1.0 (excluding 1.0).
- A more useful random number generator is provided in the java.util.Random class, as shown below:



The Random Class

- If two Random objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two Random objects with the same seed 3.

```
java.util.Random random1 = new java.util.Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
   System.out.print(random1.nextInt(1000) + " ");
java.util.Random random2 = new java.util.Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
   System.out.print(random2.nextInt(1000) + " ");
```

- The code generates the same sequence of random int values:

  From random1: 734 660 210 581 128 202 549 564 459 961

  From random2: 734 660 210 581 128 202 549 564 459 961

References

References

- Y. Daniel Liang, **Introduction to Java Programming**, Sixth Edition, Pearson Education, 2007. (Chapter 7)

- S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, M. Hoeber, **The Java Tutorial: A Short Course on the Basics**, 4th Edition, Prentice Hall, 2006. (Chapter 4)

The End