25. Generic Programming

Java

Fall 2009

Instructor: Dr. Masoud Yaghini

Outline

- Polymorphism and Generic Programming
- Casting Objects and the instanceof Operator
- The protected Data and Methods
- The final Classes, Methods, and Variables
- The this Keyword
- Abstract Methods and Classes
- References

Polymorphism and Generic Programming

Polymorphism

- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.
- A subclass is a specialization of its superclass
- **Every instance of a subclass is an instance of its superclass, but not vice versa.**
- For example, every **circle** is an object, but not every **object** is a circle.
- Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.

An Example

- Example:
  - PolymorphismDemo.java

- The output?

```
Student
Student
Person
java.lang.Object@10b30a7
```

Polymorphism

- When the method m(Object x) is executed, the argument x's toString method is invoked.
- x may be an instance of GraduateStudent, Student, Person, or Object.
- Classes GraduateStudent, Student, Person, and Object have their own implementations of the toString method.
- Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.

Polymorphism

- This capability is known as **dynamic binding** or **polymorphism** (from a Greek word meaning "many forms") because one method has many implementations.

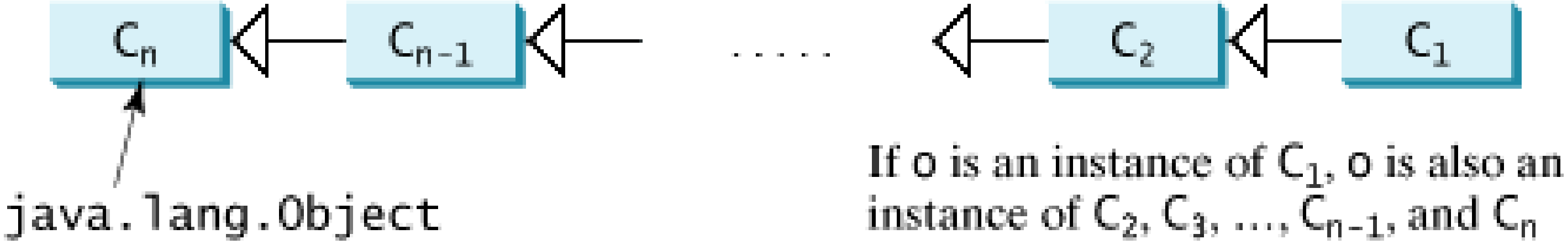
- **Polymorphism is a feature that an object of a subtype can be used wherever its supertype value is required**.

Generic Programming

- Polymorphism allows methods to be used generically for a wide range of object arguments.
- This is known as **generic programming**. If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String).
- When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object invoked (e.g., toString) is determined dynamically.

Polymorphism

- **Polymorphism** works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n
- Where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n, as shown below:



If o is an instance of C_1, o is also an instance of C_2, C_3, …, C_{n-1}, and C_n

- That is, C_n is the most general class, and C_1 is the most specific class.
- In Java, C_n is the Object class.

Polymorphism

- If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1}, and C_n, in this order, until it is found.
- Once an implementation is found, the search stops and the first-found implementation is invoked.
- For example, when m(new GraduateStudent()) is invoked, the toString method defined in the Student class is used.

Casting Objects and the instanceof Operator

Casting Objects

- You have already used the casting operator to convert variables of one primitive type to another.
- Casting can also be used to convert **an object of one class type to another** within an inheritance hierarchy.
- In the preceding section, the statement

  m(student);

  – assigns the object student to a parameter of the Object type.
- This statement is equivalent to

  Object o = new Student(); // Implicit casting m(o);

Casting Objects

- The statement

  Object o = new Student(),

  - is legal because an instance of Student is automatically an instance of Object.
  - It is known as **implicit casting**,

- Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

  Student b = o;

  - A compilation error would occur. Why?
  - Because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student.

Casting Objects

- To tell the compiler that o is a Student object, use an **explicit casting**.
- Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student) o; // Explicit casting
```

Casting Objects

- **Upcasting**
  - When casting an instance of a subclass to a variable of a superclass
  - It is possible, because an instance of a subclass is always an instance of its superclass.
- **Downcasting**
  - When casting an instance of a superclass to a variable of its subclass
  - **Explicit casting** must be used to confirm your intention to the compiler with the (SubclassName) cast notation.

instanceof Operator

- For the downcasting to be successful, you must make sure that the object to be cast is an instance of the subclass.
- If the superclass object is not an instance of the subclass, a runtime ClassCastException occurs.
- For example, if an object is not an instance of Student, it cannot be cast into a variable of Student.
- Therefore, to ensure that the object is an instance of another object before attempting a casting.
- This can be accomplished by using the **instanceof** operator.

instanceof Operator

- Consider the following code:

```
Object myObject = new Circle();
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle)
{
    myObject = (Circle) myObject;
    System.out.println("The circle diameter is " +
    myObject.getDiameter());
}
```

Casting Objects

- To help understand casting, you may also consider the analogy of fruit, apple, and orange, with the Fruit class as the superclass for Apple and Orange.
- An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit.
- However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

Casting Objects

- Why casting is necessary?
- Variable myObject is declared Object.
- The declared type decides which method to match at compile time. Using myObject.getDiameter() would cause a compilation error because the Object class does not have the getDiameter method.
- The compiler cannot find a match for myObject.getDiameter().
- It is necessary to cast myObject into the Circle type to tell the compiler that myObject is also an instance of Circle.

Casting Objects

- Why not declare myObject as a Circle type in the first place?
- To enable generic programming, it is a good practice to declare a variable with a supertype, which can accept a value of any subtype.
- Example:
  - TestPolymorphismCasting.java

TestPolymorphismCasting.java

- The program uses implicit casting to assign a Circle object to object1 and a Rectangle object to object2, and then invokes the displayObject method to display the information on these objects.
- **Casting can only be done when the source object is an instance of the target class**.
- The program uses the instanceof operator to ensure that the source object is an instance of the target class before performing a casting

TestPolymorphismCasting.java

- The object member access operator (.) precedes the casting operator.
- Use parentheses to ensure that casting is done before the . operator, as in

  ((Circle) object).getArea();

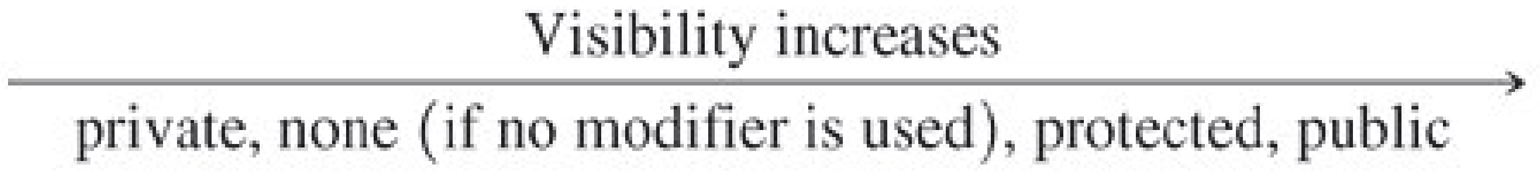
The protected Data and Methods

The protected Data and Methods

- The protected modifier can be applied on data and methods in a class.
- A protected data or a protected method in a **public class** can be accessed by any class in the **same package** or **its subclasses**, even if the subclasses are in a different package.
- The modifiers private, protected, and public are known as visibility or accessibility modifiers because they specify how class and class members are accessed.

Visibility modifiers

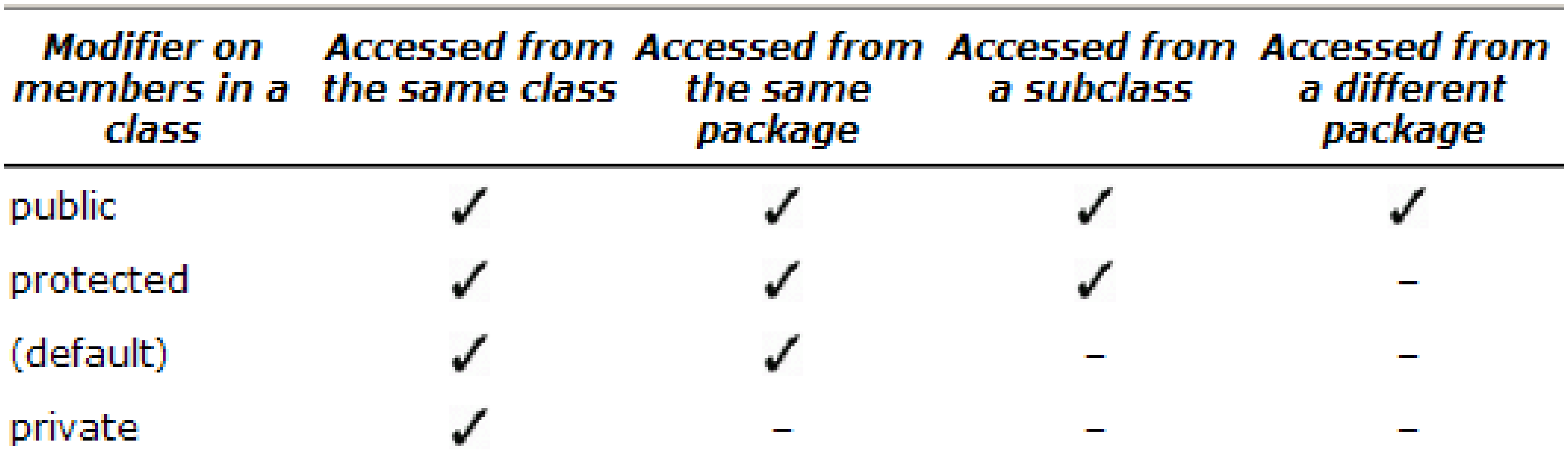
- The visibility of these modifiers increases in this order:



Visibility increases

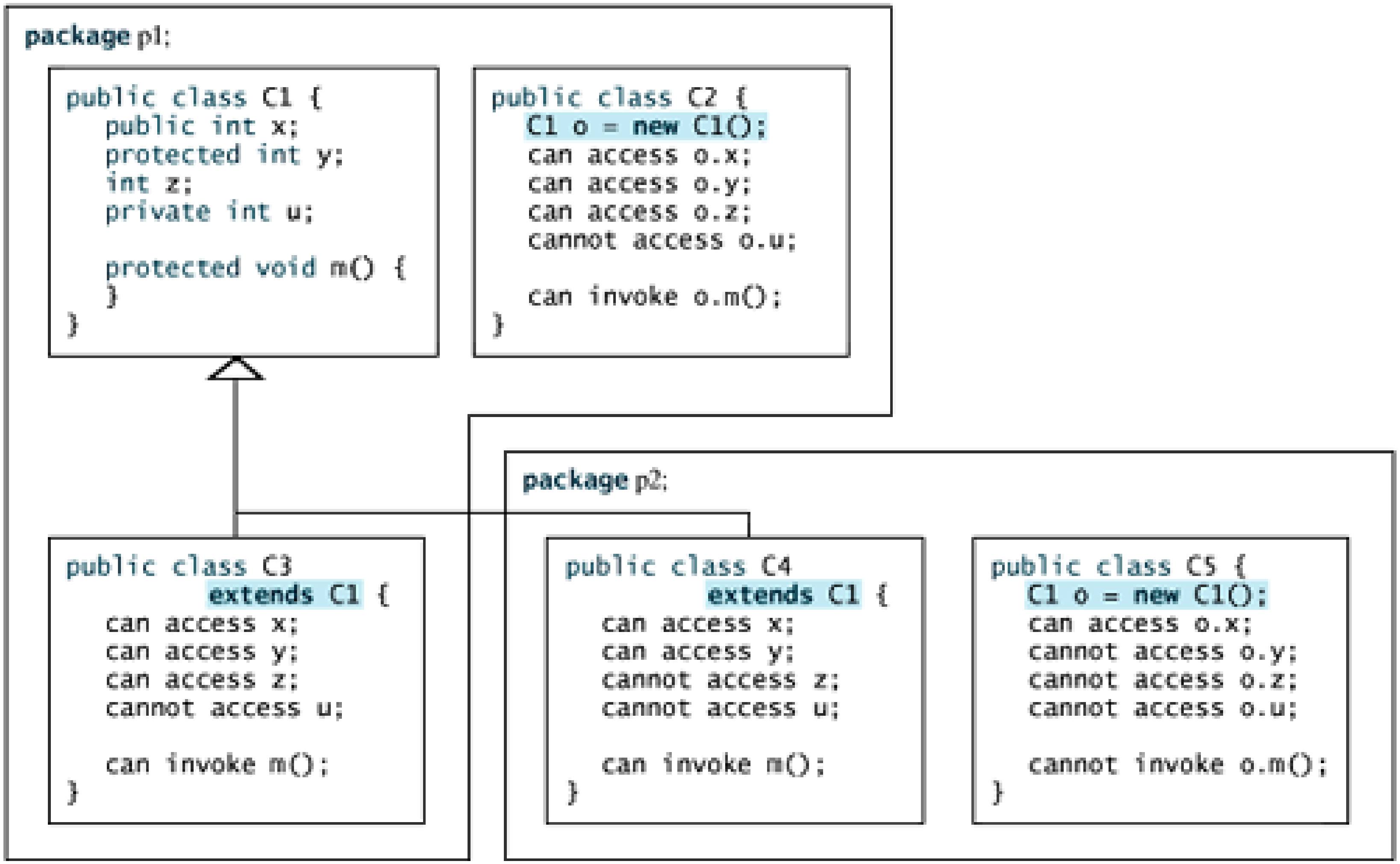
private, none (if no modifier is used), protected, public

- Summarizing the accessibility of the members in a class



Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
(default)	✓	✓	–	–
private	✓	–	–	–

Visibility modifiers



package p1;

```
public class C1 {
   public int x;
   protected int y;
   int z;
   private int u;

   protected void m() {
   }
}
```

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

   can invoke o.m();
}
```

```
public class C3
         extends C1 {
   can access x;
   can access y;
   can access z;
   cannot access u;

   can invoke m();
}
```

package p2;

```
public class C4
         extends C1 {
   can access x;
   can access y;
   cannot access z;
   cannot access u;

   can invoke m();
}
```

```
public class C5 {
   C1 o = new C1();
   can access o.x;
   cannot access o.y;
   cannot access o.z;
   cannot access o.u;

   cannot invoke o.m();
}
```

A Subclass Cannot Weaken the Accessibility

- A subclass may override a protected method in its superclass and change its visibility to public.
- However, a subclass **cannot weaken** the accessibility of a method defined in the superclass.
- For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

The final Classes, Methods, and Variables

The final Classes, Methods, and Variables

- The final class cannot be extended:

```
final class Math
{
          ...
}
```

- The final variable is a constant:

```
  final static double PI = 3.14159;
```

- The **final** method cannot be overridden by its subclasses.

The final Classes, Methods, and Variables

- The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method.
- A final local variable is a constant inside a method.

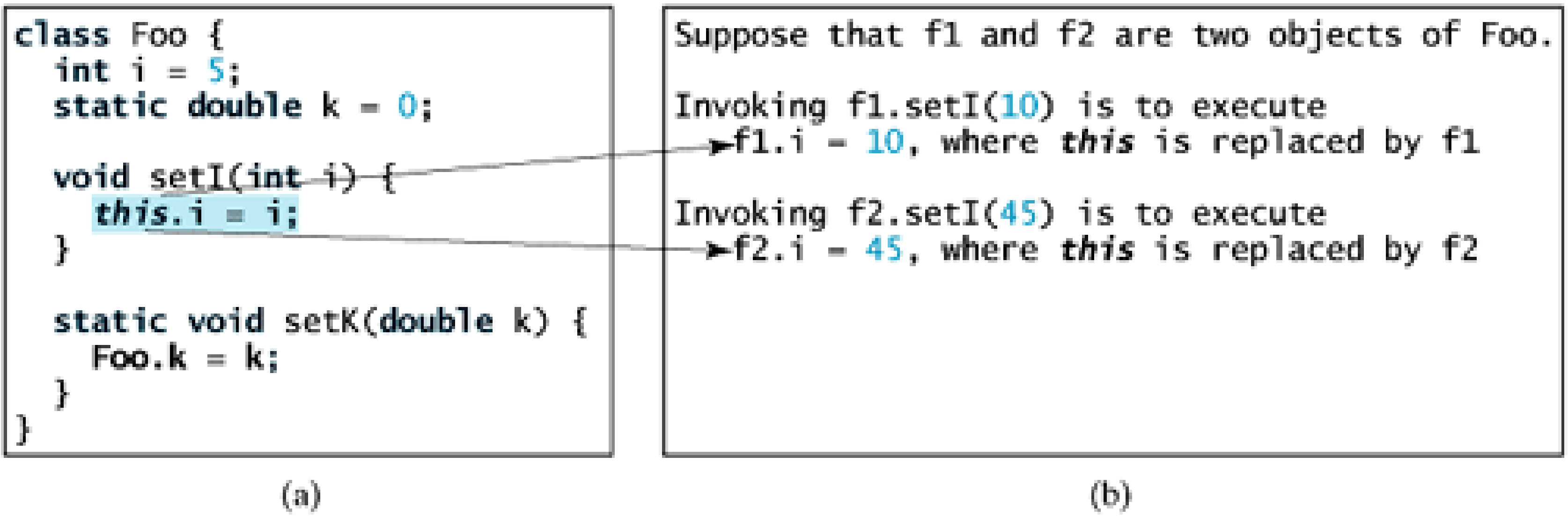
The this Keyword

The this Keyword

- A **data field name** is often used as the parameter name in a set method for the property.
- In this case, you need to reference the hidden data field name in the method in order to set a new value to it.
- A **hidden static variable** can be accessed simply by using the ClassName.StaticVariable reference.
- A **hidden instance variable** can be accessed by using the keyword this.InstanceVariable.

The this Keyword

- The keyword this serves as the proxy for the object that invokes the method.



```
class Foo {
  int i = 5;
  static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    Foo.k = k;
  }
}
```

(a)

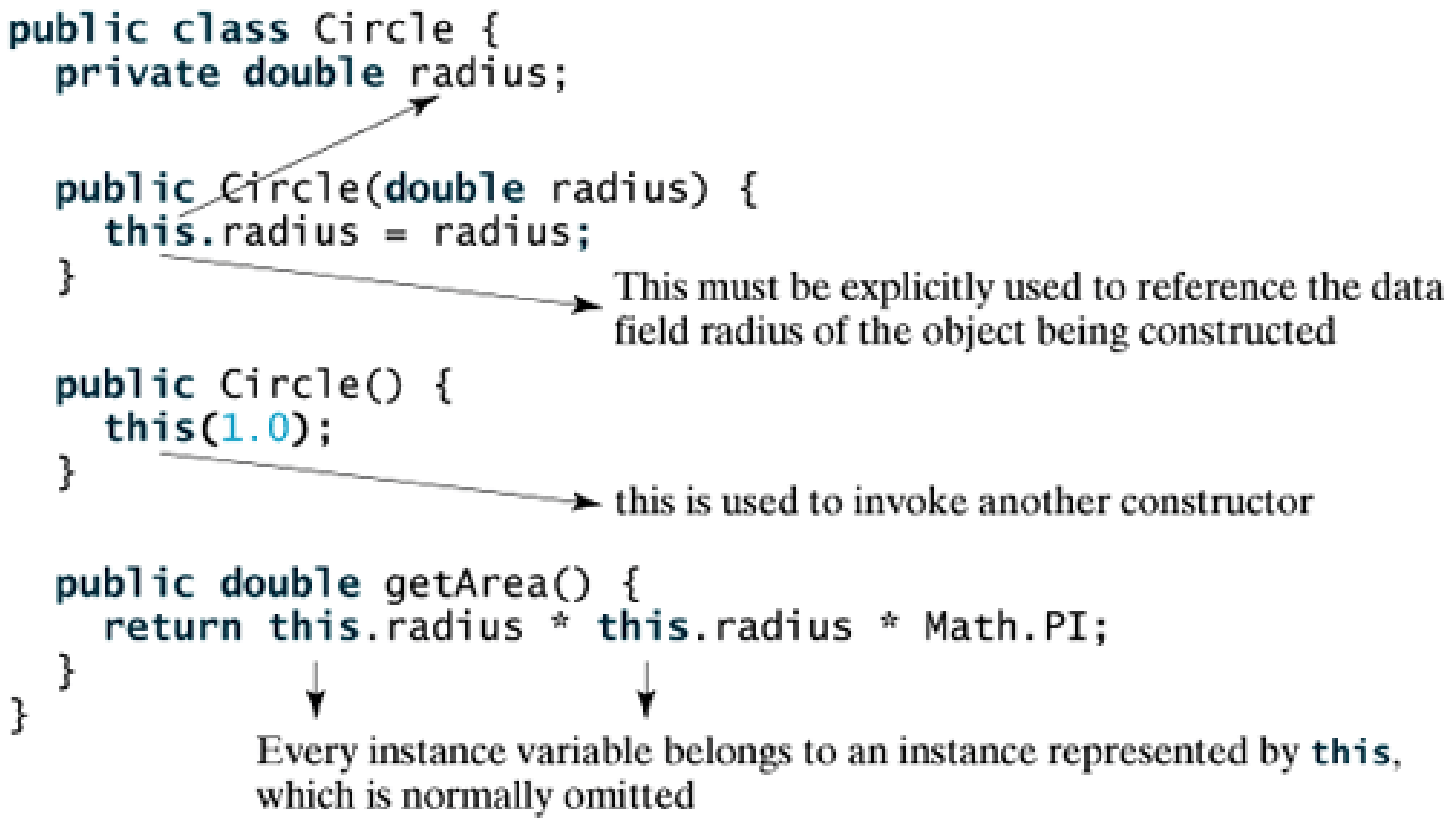
```
Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
  f1.i = 10, where this is replaced by f1

Invoking f2.setI(45) is to execute
  f2.i = 45, where this is replaced by f2
```

(b)

- The line this.i = i means "assign the value of parameter i to the data field i of the calling object."

The this Keyword

- The keyword this can also be used inside a constructor to invoke another constructor of the same class.



```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

This must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

Abstract Methods and Classes

The abstract Modifier

- The abstract method
  - Method signature without implementation
  - Its implementation is provided by the subclasses.
- The abstract class
  - A class that contains abstract methods must be declared abstract.
  - Cannot be **instantiated** (you cannot create instances of abstract classes)

Abstract Classes

- In the preceding chapter we compute areas and perimeters for all geometric objects
- It is better to **declare** the getArea() and getPerimeter() methods in the GeometricObject class.
- These methods cannot be implemented in the GeometricObject class because their implementation is dependent on the specific type of geometric object.
- Such methods are referred to as **abstract methods**.

An Example

- Example:
  - GeometricObject.java
  - Circle.java
  - Rectangle.java
  - TestAbstractClass.java
- Output:

```
The two objects have the same area? false

The area is 78.53981633974483
The perimeter is 31.41592653589793

The area is 15.0
The perimeter is 16.0
```

Abstract Classes

- An abstract class cannot be instantiated using the new operator
- But you can still define its constructors, which are invoked in the constructors of its subclasses.
- For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

Abstract Classes

- A class that contains abstract methods must be abstract.
- However, it is possible to declare an abstract class that contains no abstract methods.
- In this case, you cannot create instances of the class using the new operator.
- This class is used as a base class for defining a new subclass.

References

References

- Y. Daniel Liang, **Introduction to Java Programming**, Sixth Edition,
  Pearson Education, 2007. (Chapter 9 & 10)

The End