



Revision on Basic Java Programming

Loops

Motivation:

Suppose that you need to print a string (e.g., "Welcome to Java!") a hundred times. It would be tedious to have to write the following statement a hundred times:

```
System.out.println("Welcome to Java!");
```

So, how do you solve this problem?



Opening Problem

Problem:

100
times

```
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");
```

...

...

...

```
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");
```



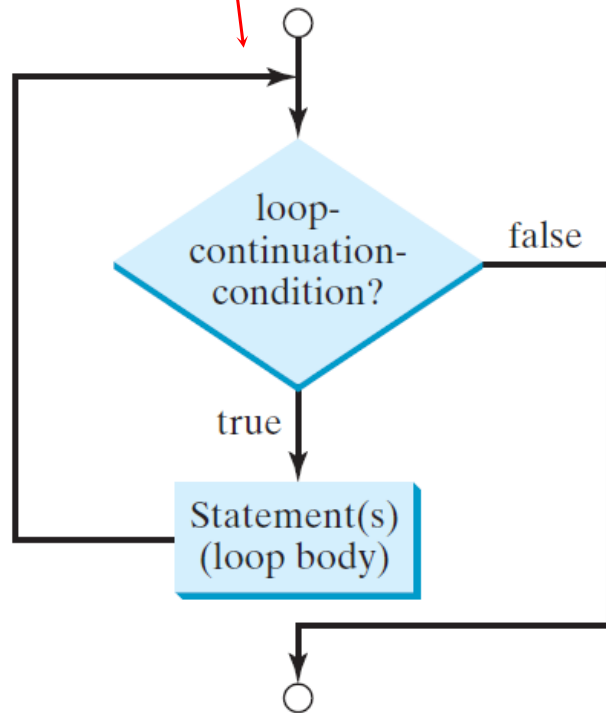
Introducing while Loops

```
int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java");
    count++;
}
```

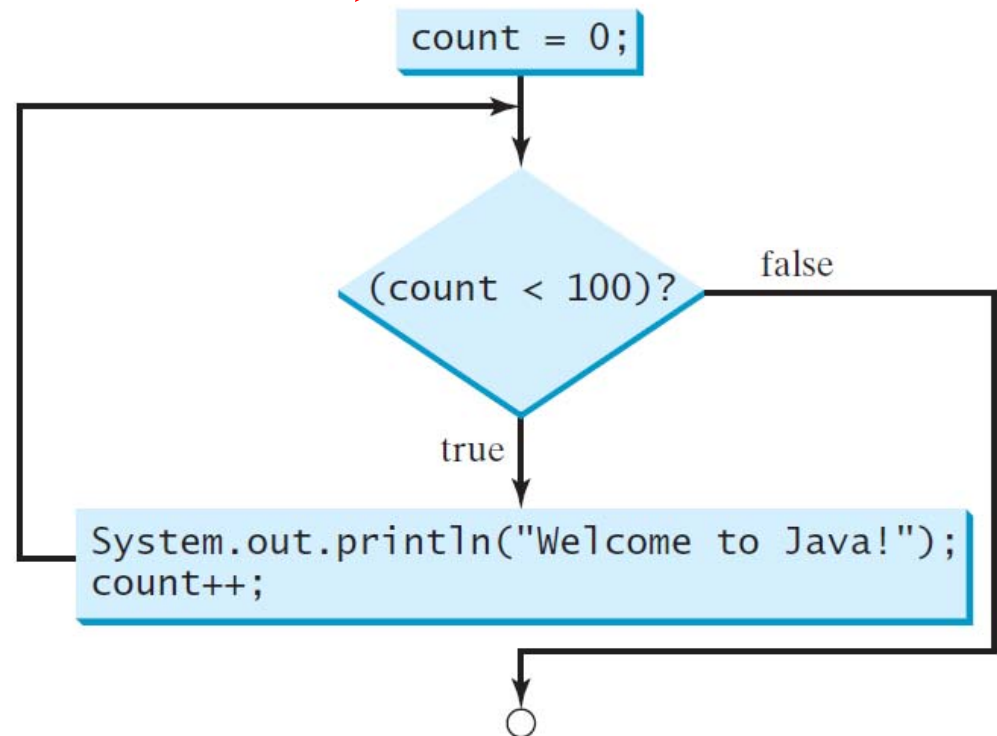


while Loop Flow Chart

```
while (loop-continuation-condition) {  
    // loop-body;  
    Statement(s);  
}
```

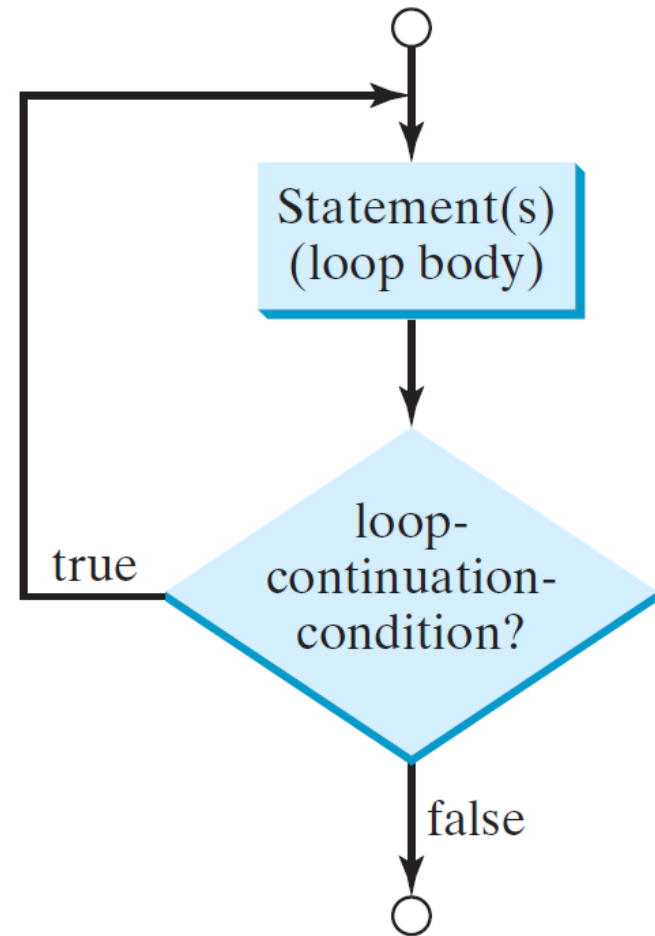


```
int count = 0;  
while (count < 100) {  
    System.out.println("Welcome to Java!");  
    count++;  
}
```



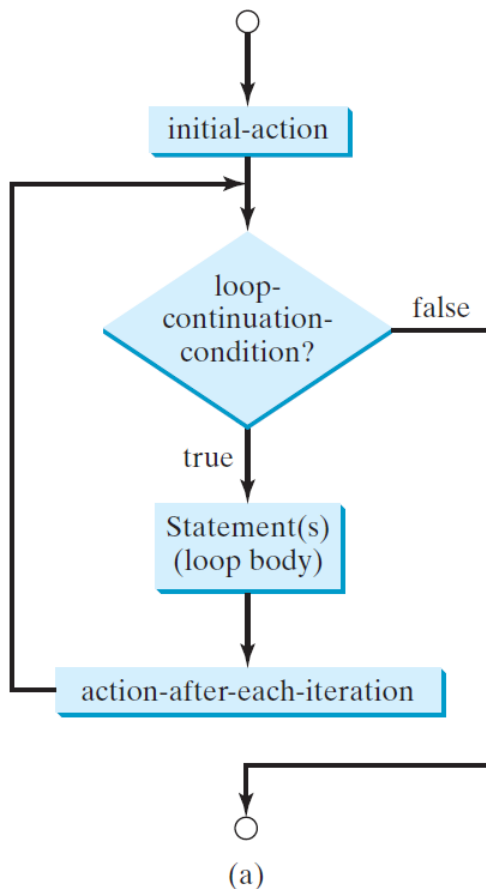
do-while Loop

```
do {  
    // Loop body;  
    Statement(s);  
} while (loop-continuation-condition);
```

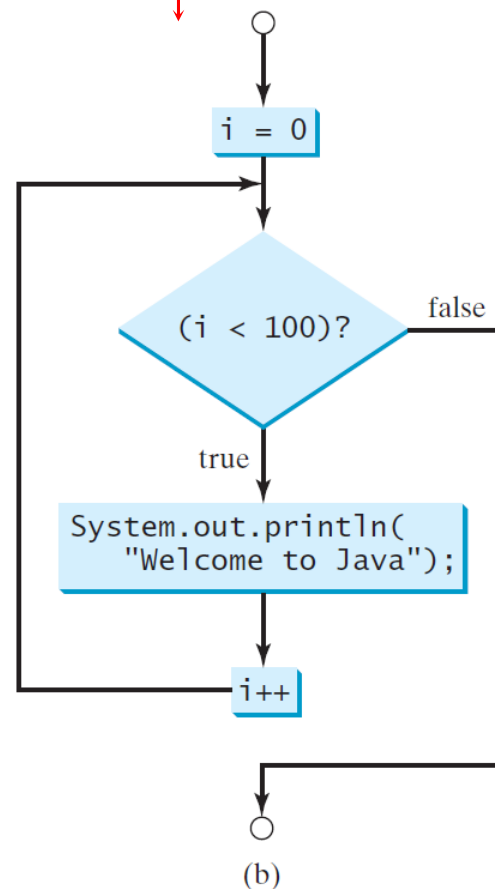


for Loops

```
for (initial-action; loop-  
    continuation-condition; action-  
    after-each-iteration) {  
    // loop body;  
    Statement(s);  
}
```



```
int i;  
for (i = 0; i < 100; i++) {  
    System.out.println(  
        "Welcome to Java!");  
}
```



Note

The initial-action in a for loop can be a list of zero or more comma-separated expressions. The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements. Therefore, the following two for loops are correct. They are rarely used in practice, however.

```
for (int i = 1; i < 100; System.out.println(i++));
```

```
for (int i = 0, j = 0; (i + j < 10); i++, j++) {
```

```
// Do something
```

```
}
```



Note

If the loop-continuation-condition in a for loop is omitted, it is implicitly true. Thus the statement given below in (a), which is an infinite loop, is correct. Nevertheless, it is better to use the equivalent loop in (b) to avoid confusion:

```
for ( ; ; ) {  
    // Do something  
}
```

(a)

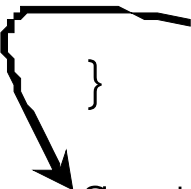
Equivalent

```
while (true) {  
    // Do something  
}
```

(b)

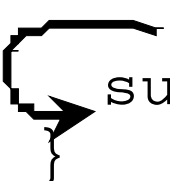
break

```
public class TestBreak {  
    public static void main(String[] args) {  
        int sum = 0;  
        int number = 0;  
  
        while (number < 20) {  
            number++;  
            sum += number;  
            if (sum >= 100)  
                break;  
        }  
        System.out.println("The number is " + number);  
        System.out.println("The sum is " + sum);  
    }  
}
```

A black arrow originates from the 'break;' statement on line 12 and points to the closing curly brace of the 'while' loop on line 11, illustrating that the break statement exits the loop immediately.

continue

```
public class TestContinue {  
    public static void main(String[] args) {  
        int sum = 0;  
        int number = 0;  
  
        while (number < 20) {  
            number++;  
            if (number == 10 || number == 11)  
                continue;  
            sum += number;  
        }  
  
        System.out.println("The sum is " + sum);  
    }  
}
```





Methods

Defining Methods

A method is a collection of statements that are grouped together to perform an operation.

Define a method

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

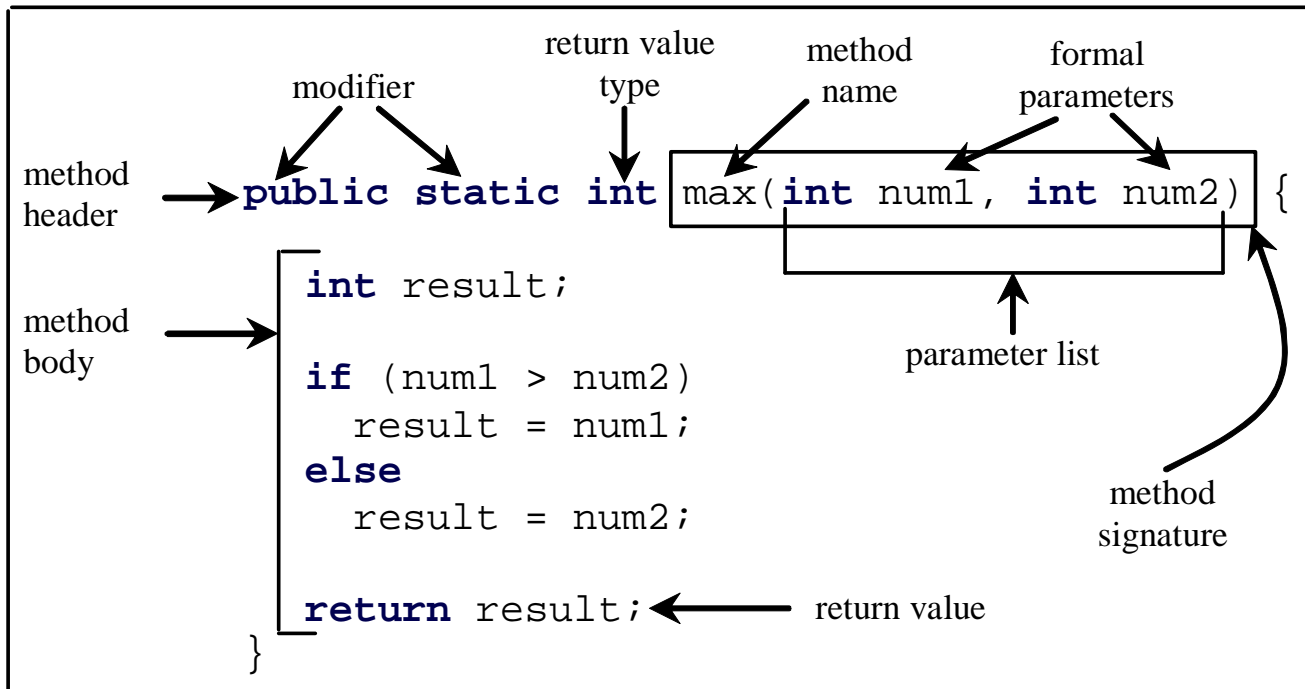
Invoke a method

```
int z = max(x, y);  
        ↑   ↑  
    actual parameters  
    (arguments)
```

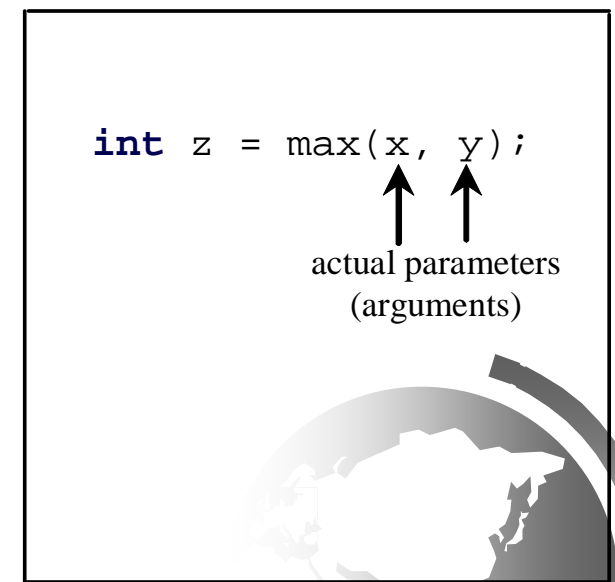
Defining Methods

A method is a collection of statements that are grouped together to perform an operation.

Define a method



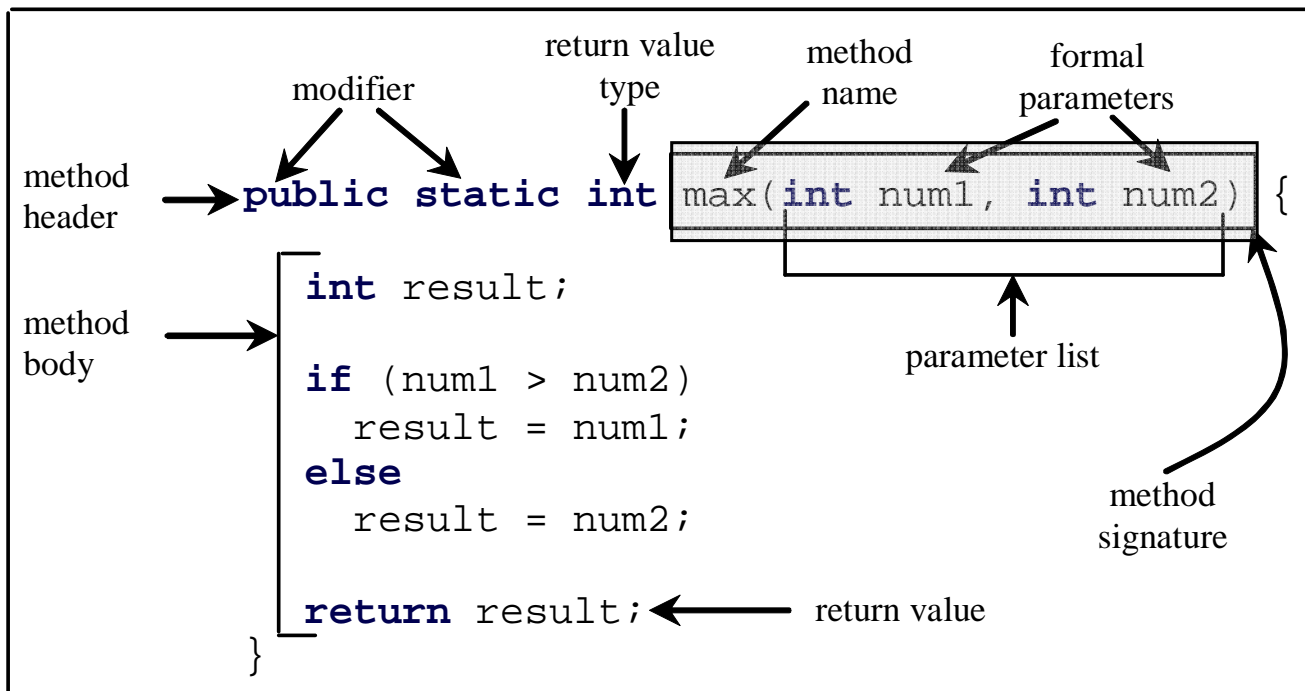
Invoke a method



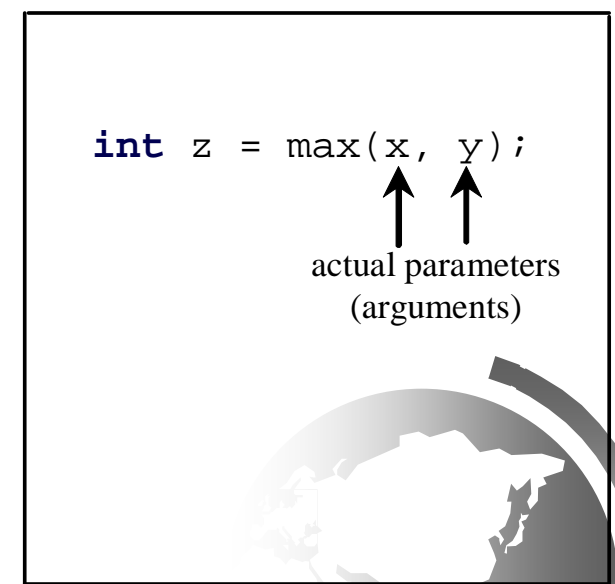
Method Signature

Method signature is the combination of the method name and the parameter list.

Define a method



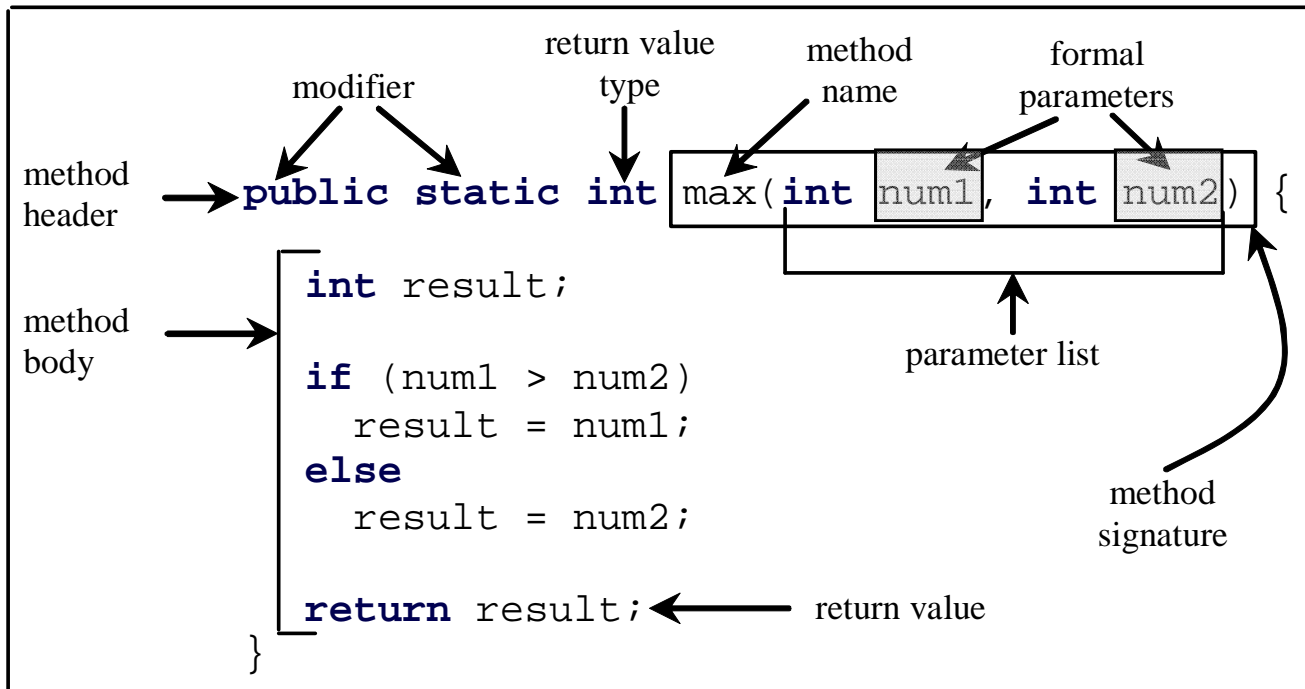
Invoke a method



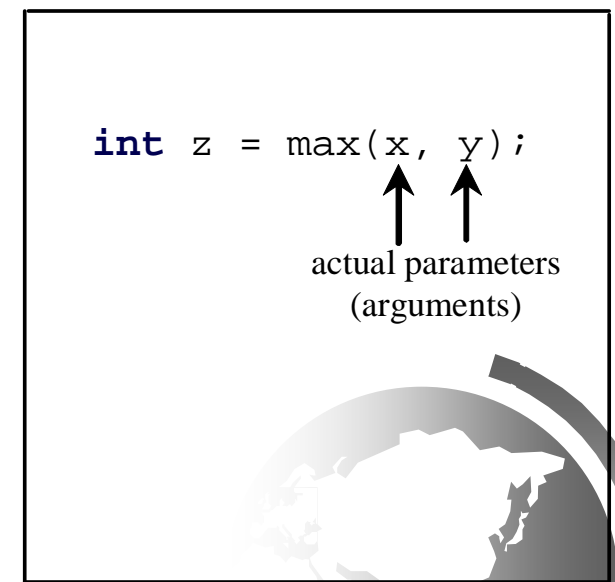
Formal Parameters

The variables defined in the method header are known as *formal parameters*.

Define a method



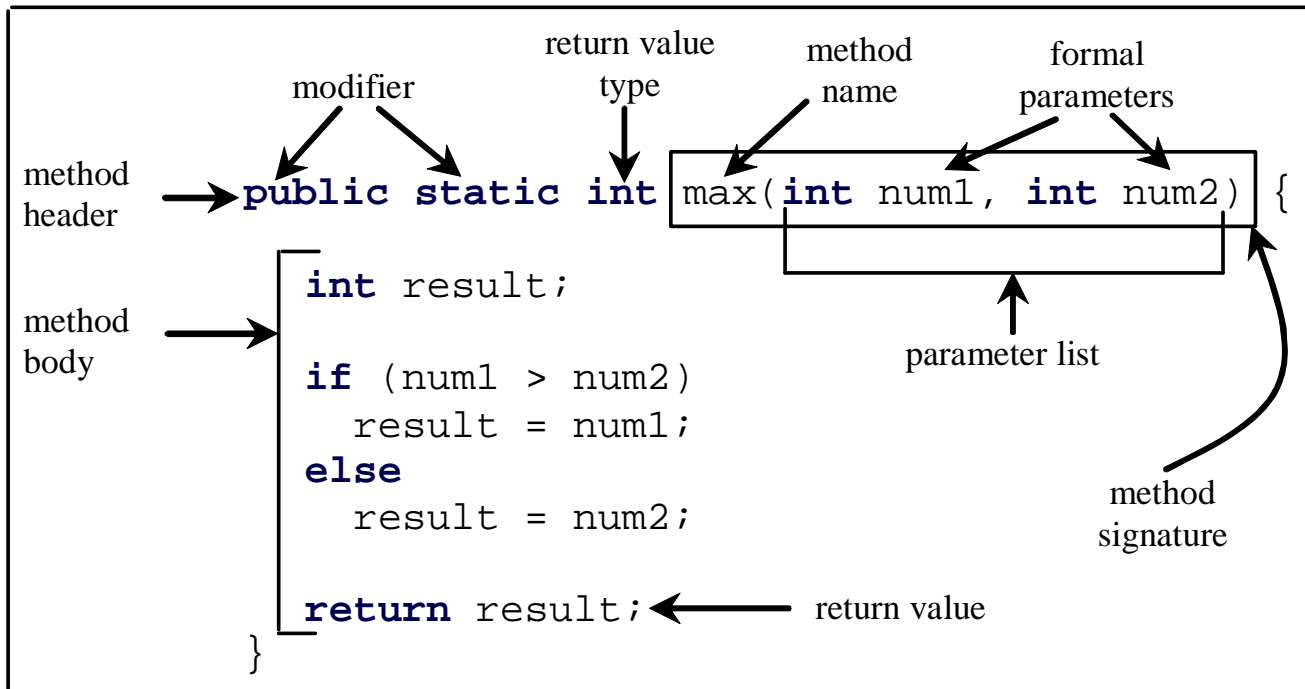
Invoke a method



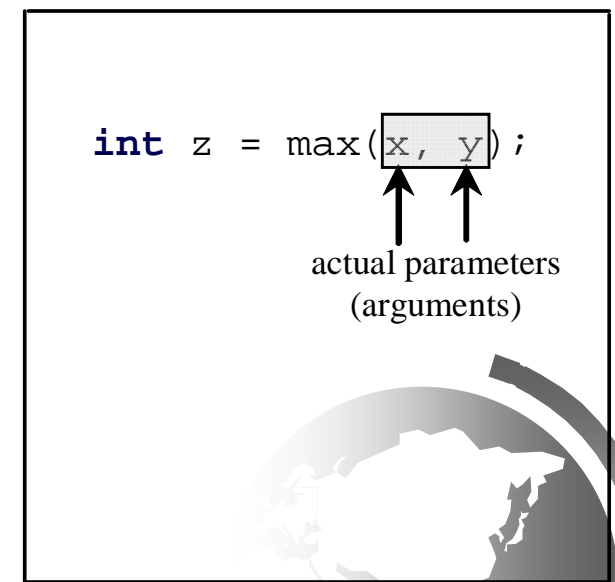
Actual Parameters

When a method is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*.

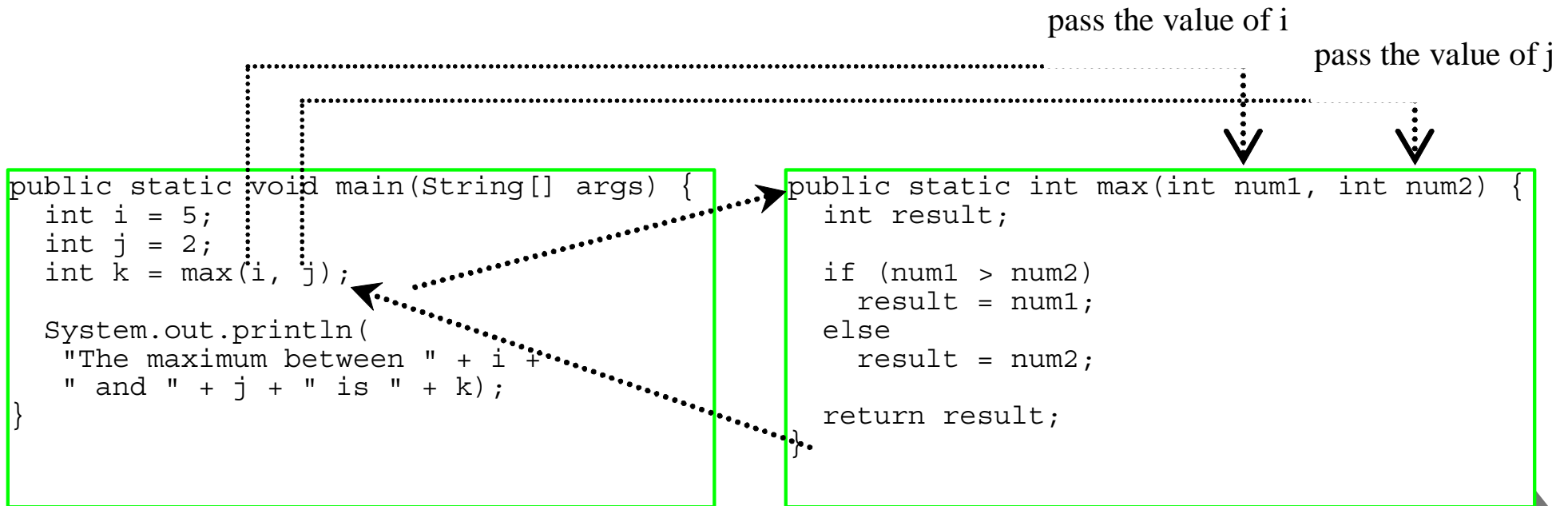
Define a method



Invoke a method



Calling Methods



CAUTION

A return statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.

```
public static int sign(int n) {  
    if (n > 0)  
        return 1;  
    else if (n == 0)  
        return 0;  
    else if (n < 0)  
        return -1;  
}
```

(a)

Should be

```
public static int sign(int n) {  
    if (n > 0)  
        return 1;  
    else if (n == 0)  
        return 0;  
    else  
        return -1;  
}
```

(b)

To fix this problem, delete if (n < 0) in (a), so that the compiler will see a return statement to be reached regardless of how the if statement is evaluated.

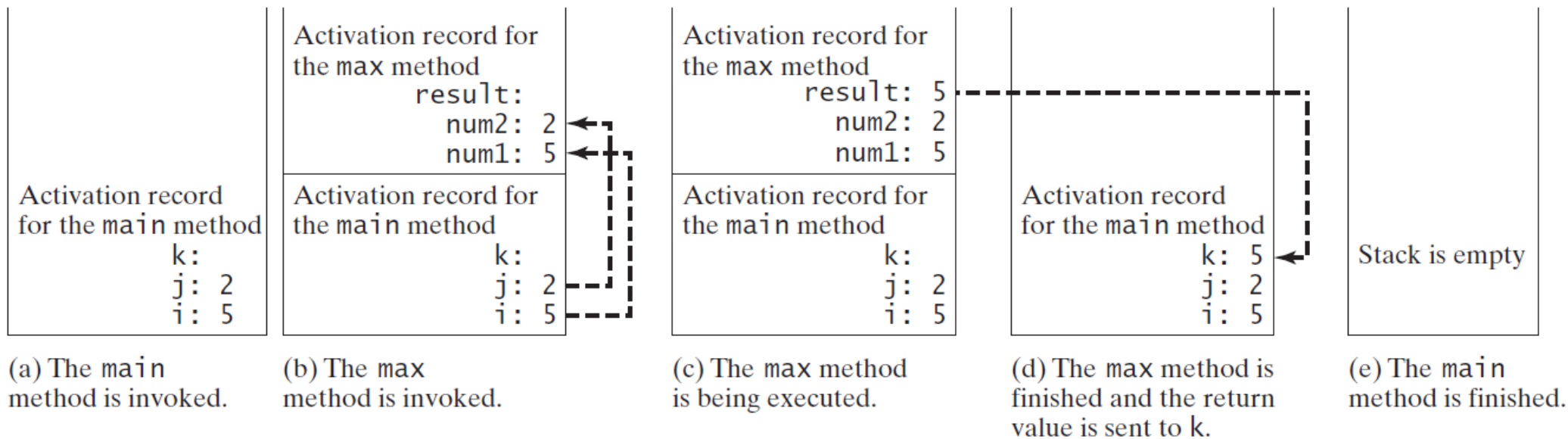
Call Stacks

```
public static void main(String[] args) {  
    int i = 5;  
    int j = 2;  
    int k = max(i, j);  
  
    System.out.println(  
        "The maximum between " + i +  
        " and " + j + " is " + k);  
}
```

```
public static int max(int num1, int num2) {  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```



Call Stacks



Overloading Methods

Overloading the max Method

```
public static double max(double num1, double  
    num2) {  
    if (num1 > num2)  
        return num1;  
    else  
        return num2;  
}
```

TestMethodOverloading

Run



Scope of Local Variables

A local variable: a variable defined inside a method.

Scope: the part of the program where the variable can be referenced.

The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.



Scope of Local Variables, cont.

You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.



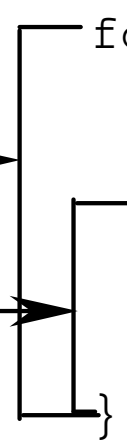
Scope of Local Variables, cont.

A variable declared in the initial action part of a for loop header has its scope in the entire loop. But a variable declared inside a for loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable.

```
public static void method1() {  
    .  
    .  
    for (int i = 1; i < 10; i++) {  
        .  
        .  
        int j;  
        .  
        .  
        .  
    }  
}
```

The scope of i →

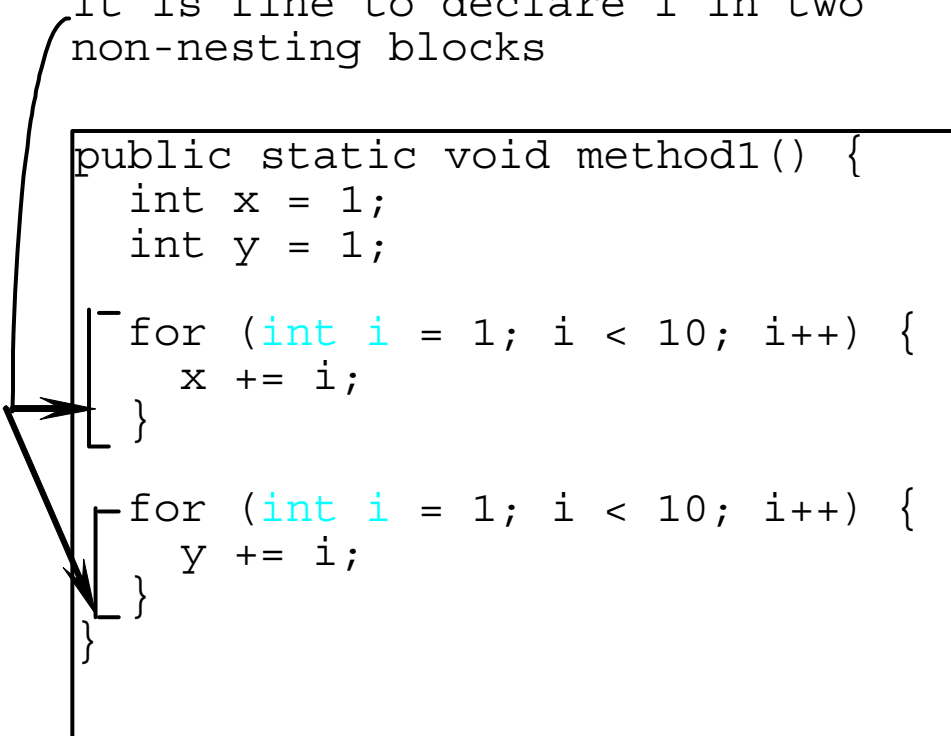
The scope of j →



Scope of Local Variables, cont.

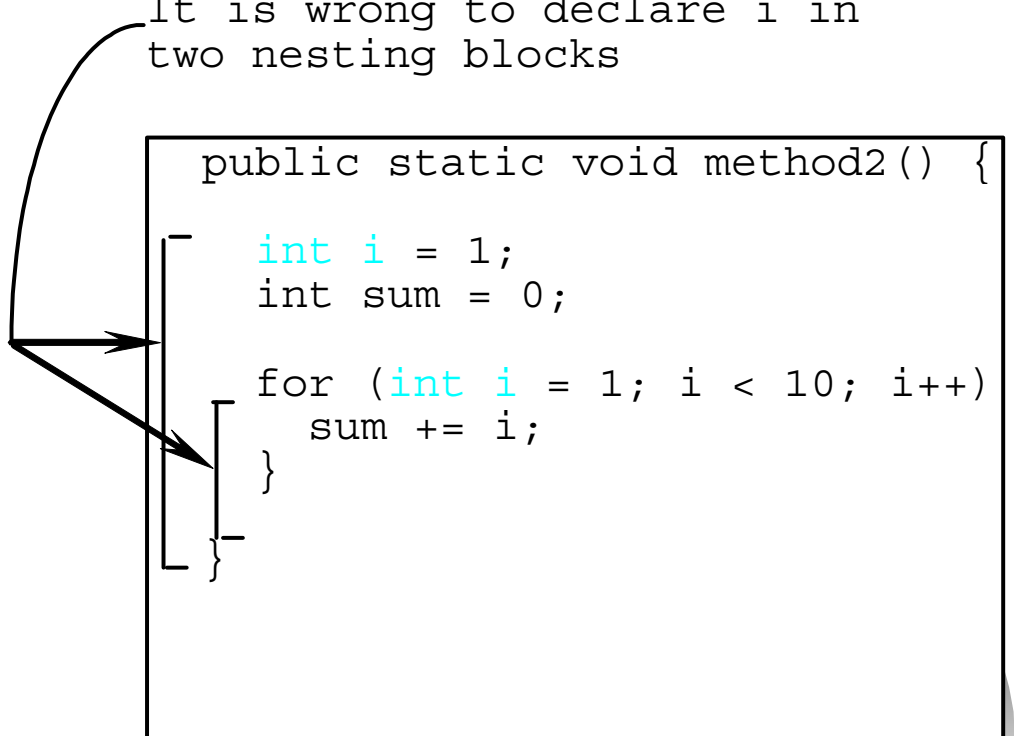
It is fine to declare `i` in two non-nesting blocks

```
public static void method1() {  
    int x = 1;  
    int y = 1;  
  
    for (int i = 1; i < 10; i++) {  
        x += i;  
    }  
  
    for (int i = 1; i < 10; i++) {  
        y += i;  
    }  
}
```



It is wrong to declare `i` in two nesting blocks

```
public static void method2() {  
    int i = 1;  
    int sum = 0;  
  
    for (int i = 1; i < 10; i++)  
        sum += i;  
}
```



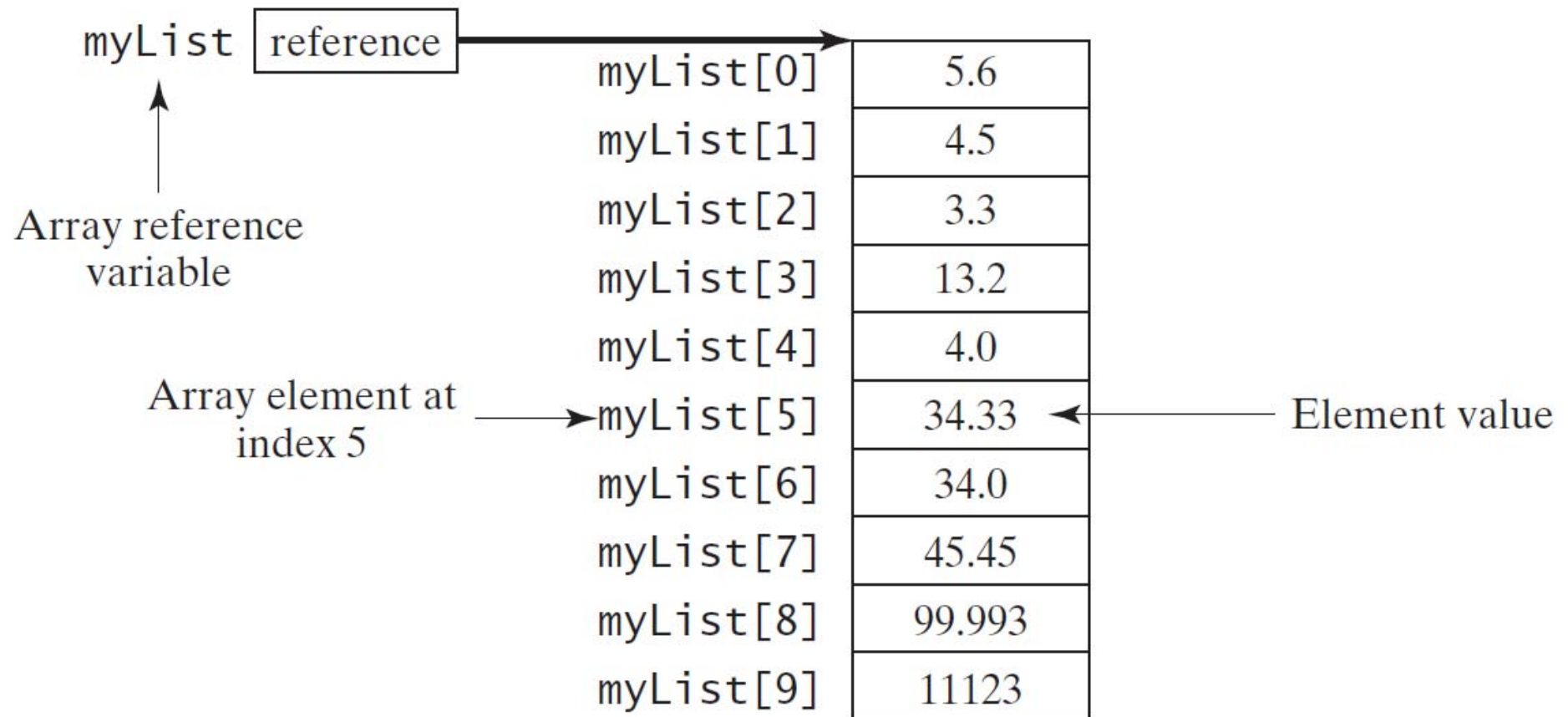


Arrays

Introducing Arrays

Array is a data structure that represents a collection of the same types of data.

```
double[] myList = new double[10];
```



Declaring Array Variables

☞ `datatype[] arrayRefVar;`

Example:

```
double[] myList;
```

```
myList = new double[10];
```

☞ `datatype arrayRefVar[];` // This style is allowed, but not preferred

Example:

```
double myList[];
```



Declaring and Creating in One Step

☞ `datatype[] arrayRefVar = new
datatype[arraySize];`

`double[] myList = new double[10];`

☞ `datatype arrayRefVar[] = new
datatype[arraySize];`

`double myList[] = new double[10];`



The Length of an Array

Once an array is created, its size is fixed. It cannot be changed. You can find its size using

```
arrayRefVar.length
```

For example,

```
myList.length returns 10
```



Indexed Variables

The array elements are accessed through the index. The array indices are *0-based*, i.e., it starts from 0 to `arrayRefVar.length-1`. In the example in Figure 6.1, `myList` holds ten double values and the indices are from 0 to 9.

Each element in the array is represented using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```



Initializing arrays with input values

```
java.util.Scanner input = new java.util.Scanner(System.in);  
System.out.print("Enter " + myList.length + " values: ");  
for (int i = 0; i < myList.length; i++)  
    myList[i] = input.nextDouble();
```



Initializing arrays with random values

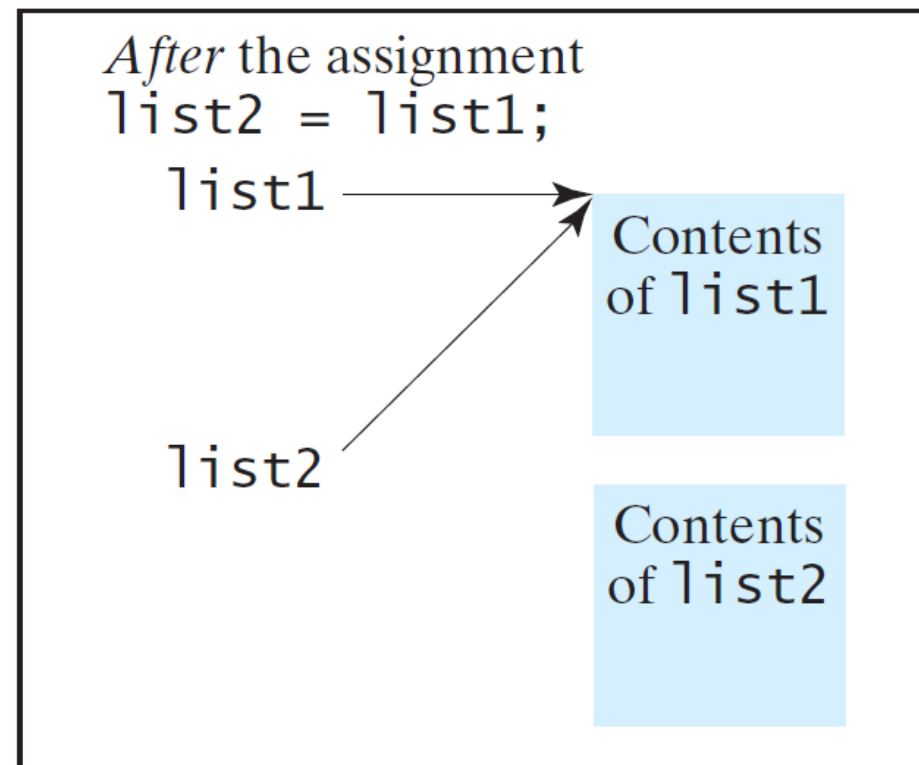
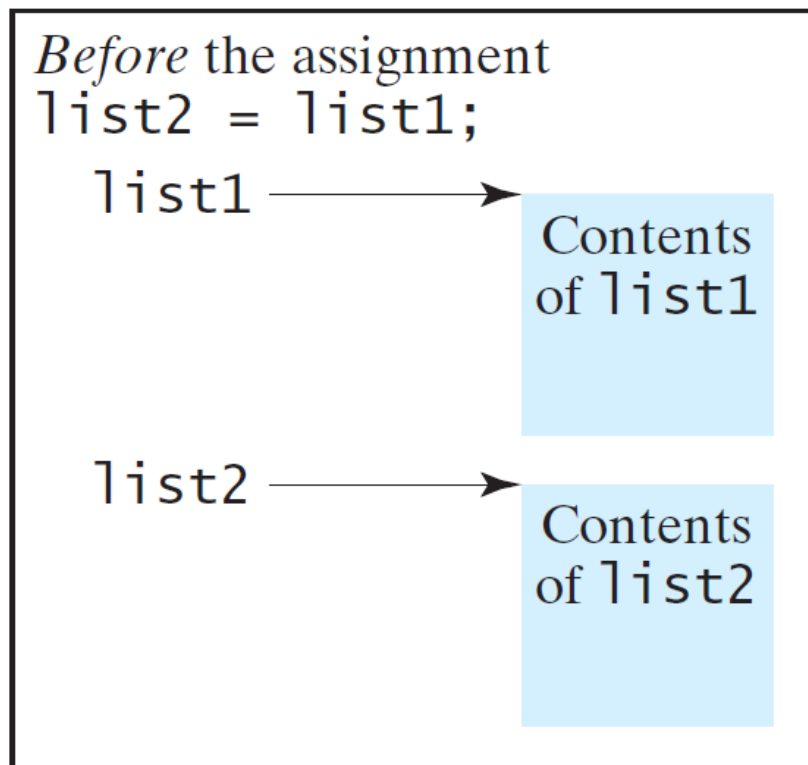
```
for (int i = 0; i < myList.length; i++) {  
    myList[i] = Math.random() * 100;  
}
```



Copying Arrays

Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (=), as follows:

```
list2 = list1;
```



Copying Arrays

Using a loop:

```
int[] sourceArray = {2, 3, 1, 5, 10};  
int[] targetArray = new  
    int[sourceArray.length];  
  
for (int i = 0; i < sourceArray.length; i++)  
    targetArray[i] = sourceArray[i];
```



Classes

Classes are constructs that define objects of the same type. A Java class uses variables to define data fields and methods to define behaviors. Additionally, a class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.



Classes

```
class Circle {  
    /** The radius of this circle */  
    double radius = 1.0;  
  
    /** 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

← Method



Constructors

```
Circle() {  
}
```

Constructors are a special kind of methods that are invoked to construct objects.

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

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.



Default Constructor

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



Declaring/Creating Objects in a Single Step


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

Assign object reference

Create an object

Example:

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



Accessing Object's Members

- ❑ Referencing the object's data:

`objectRefVar.data`

e.g., `myCircle.radius`

- ❑ Invoking the object's method:

`objectRefVar.methodName (arguments)`

e.g., `myCircle.getArea()`



Caution

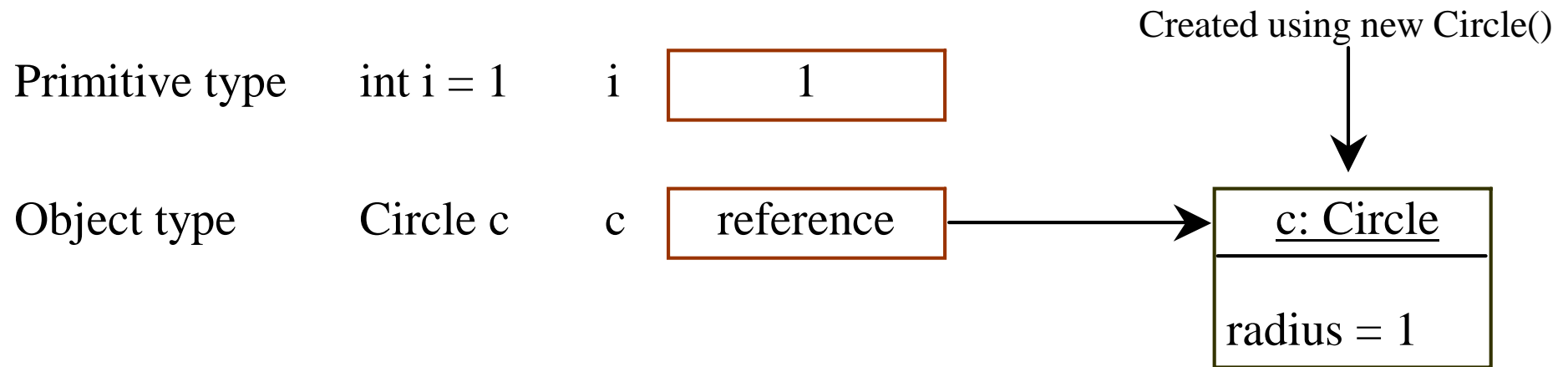
Java assigns no default value to a local variable inside a method.

```
public 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 " + x);  
        System.out.println("y is " + y);  
    }  
}
```

Compile error: variable not
initialized



Differences between Variables of Primitive Data Types and Object Types



Static Variables, Constants, and Methods

Static variables are shared by all the instances of the class.

Static methods are not tied to a specific object.

Static constants are final variables shared by all the instances of the class.



Visibility Modifiers and Accessor/Mutator Methods

By default, the class, variable, or method can be accessed by any class in the same package.

- ❑ `public`

The class, data, or method is visible to any class in any package.

- ❑ `private`

The data or methods can be accessed only by the declaring class.

The get and set methods are used to read and modify private properties.

Class Relationships

Composition

Inheritance (Chapter 13)

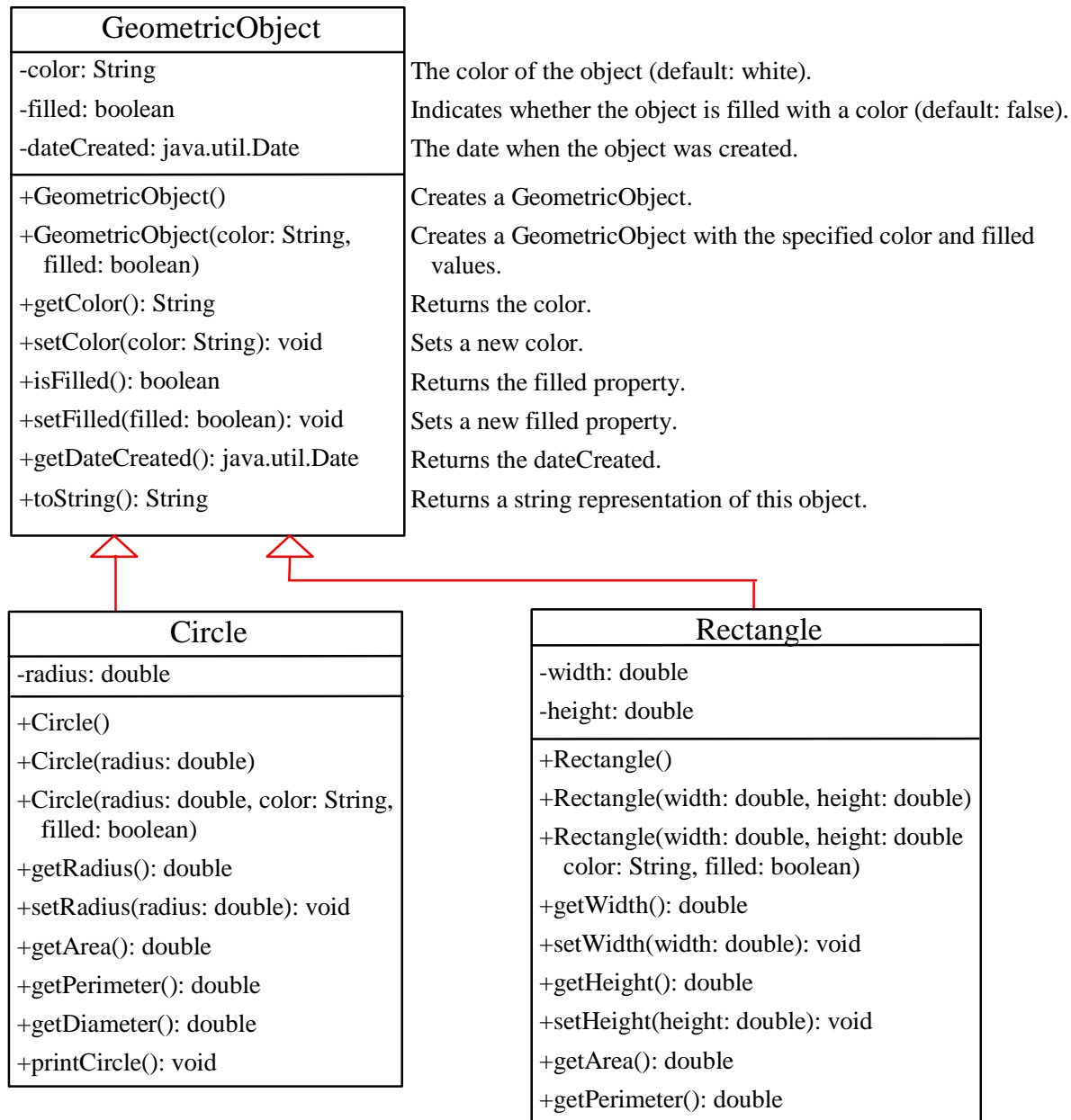
Composition represents an ownership relationship between two objects. The Child (owned) class is usually represented as a data field in the Parent (owner) class.

```
public class Name {  
    ...  
}
```

```
public class Student {  
    private Name name;  
    private Address address;  
  
    ...  
}
```

```
public class Address {  
    ...  
}
```

Inheritance: Superclasses and Subclasses



Defining a Subclass

A subclass inherits from a superclass. You can also:

- ➡ Add new properties
- ➡ Add new methods
- ➡ Override the methods of the superclass (you must override each abstract method)



Are superclass's Constructor Inherited?

No. They are not inherited.

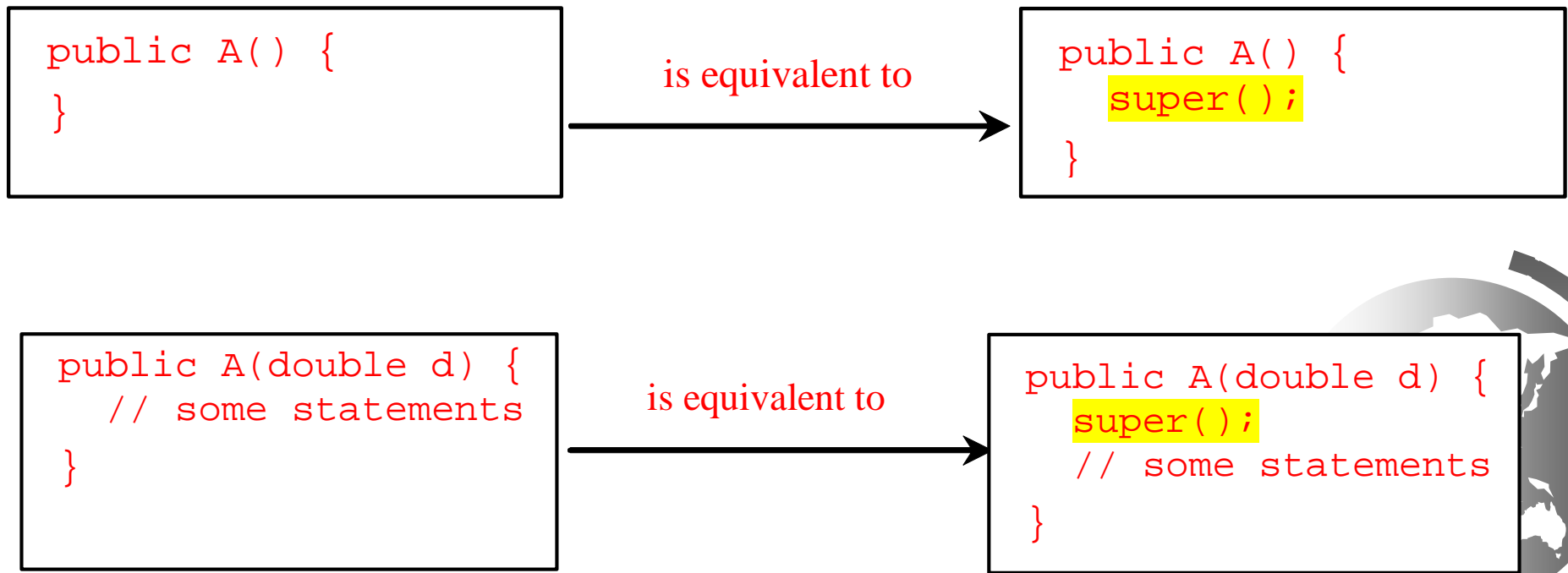
They are invoked explicitly or implicitly.

Explicitly using the `super` keyword.

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword `super`. *If the keyword `super` is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,



Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- ➡ To call a superclass constructor
- ➡ To call a superclass method



Overriding vs. Overloading

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

Polymorphism

```
class Person {
    public String toString() {
        return "Person";
    }
}

class Student extends Person {
    @Override
    public String toString() {
        return "Student";
    }
}

public class PolymorphismDemo {
    public static void main(String[] args) {

        //Polymorphism
        Person[] P = new Person[10];

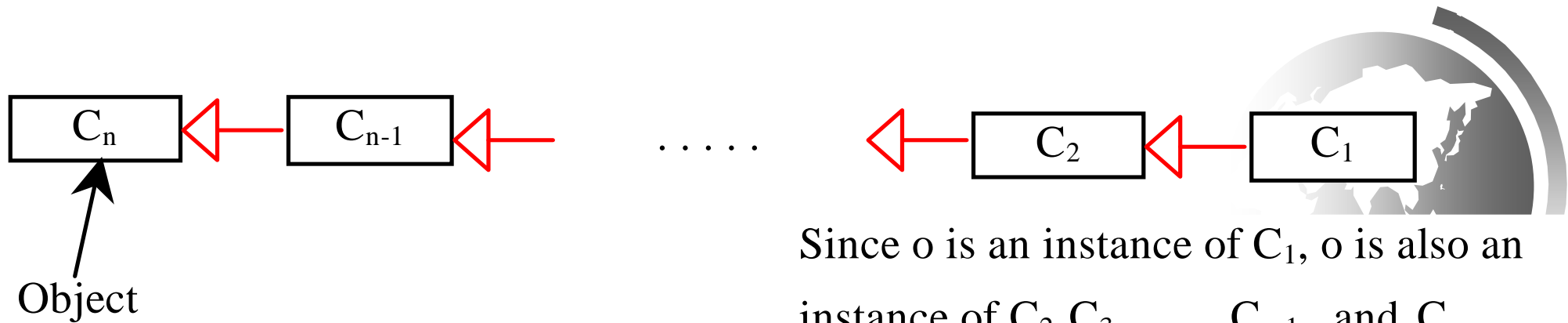
        Person m = new Person();
        Student s = new Student();
        P[0] = m;
        P[1] = s;
        System.out.println(P[0].toString());
        System.out.println(P[1].toString());
    }
}
```

Polymorphism means that a variable of a supertype can refer to a subtype object. In other words, An object of a subtype can be used wherever its supertype value is required



Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, \dots, 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 . That is, C_n is the most general class, and C_1 is the most specific class. Suppose object o is of type C_1 . If o invokes a method p , the JVM searches the implementation for the method p in C_1, C_2, \dots, 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.



Since o is an instance of C_1 , o is also an instance of C_2, C_3, \dots, C_{n-1} , and C_n

The protected Modifier

- 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.
- private, default, protected, public

Visibility increases
—————→
private, none (if no modifier is used), protected, public



Accessibility Summary

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	✓	–	–	–



The `final` Modifier

- ➡ 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.

