Object-oriented Programming

Reference:

Chapter 6 of A Programmer's Guide to Java Certification: A Comprehensive Primer.

Overview

- The inheritance relationship: *is-a*
- The aggregation relationship: *has-a*
- Overridden and overloaded methods
- The keyword `super`
- Variable shadowing
- Constructors and constructor chaining using `this()` and `super()`
- Single implementation inheritance, multiple interface inheritance and supertypes
- Assigning, casting and passing references
- The `instanceof` operator
- Polymorphism and dynamic method lookup
- Encapsulation
- Choosing between inheritance and aggregation

Extensibility by Linear Implementation Inheritance

- One fundamental mechanism for *code reuse.*
- The new class *inherits* all the members of the old class - not to be confused with *accessibility* of superclass members.
- A class in Java can only extend one other class, i.e. it can only have one *immediate* superclass.
- The superclass is specified using the `extends` clause in the header of the subclass.
- The definition of the subclass only specifies the *additional new and modified* members in its class definition.
- All classes extend the `java.lang.Object` class.

Example 1 Extending Classes

```
class Light {                       // (1)
    // Instance variables
    private int noOfWatts;          // wattage
    private boolean indicator;      // on or off
    private String location;        // placement

    // Static variable
    private static int counter;     // no. of Light objects created

    // Constructor
    Light() {
        noOfWatts = 50;
        indicator = true;
        location = new String("X");
    }

    // Instance methods
    public void switchOn()  { indicator = true; }
    public void switchOff() { indicator = false; }
    public boolean isOn() { return indicator; }

    // Static methods
    public static void writeCount() {
         System.out.println("Number of lights: " + counter);
    }
    //...
}
```

```
class TubeLight extends Light {     // (2)
    // Instance variables
    private int tubeLength;
    private int color;

    // Instance method
    public int getTubeLength() { return tubeLength; }
    // ...
}
```


Implementation Inheritance Hierarchy

- Inheritance defines the relationship *is-a* (also called *superclass–subclass* relationship) between a superclass and its subclasses.
- Classes higher up in the hierarchy are more *generalized*, as they abstract the class behavior.
- Classes lower down in the hierarchy are more *specialized*, as they customize the inherited behavior by additional properties and behavior.
- The `Object` class is always the root of any inheritance hierarchy.

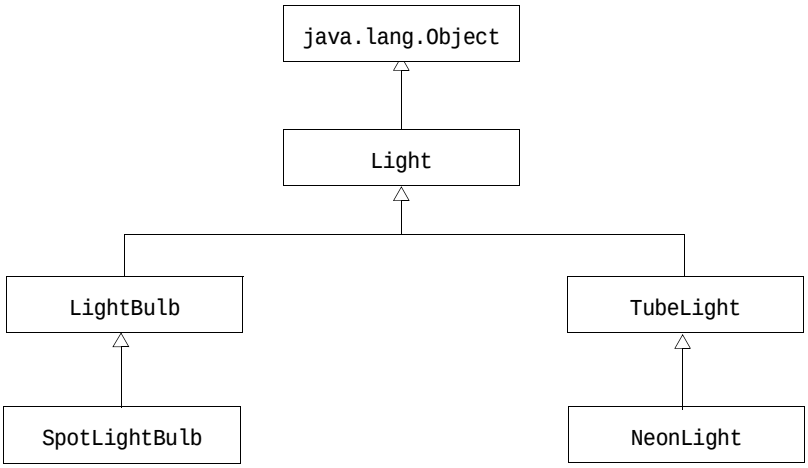


Figure 1 Inheritance Hierarchy

Implications of Inheritance

- *An object of a subclass can be used wherever an object of the superclass can be used.*
- An object of the `TubeLight` class can be used wherever an object of the superclass `Light` can be used.
  - An object of the `TubeLight` class *is-a* object of the superclass `Light`.
- The inheritance relationship is *transitive*: if class `B` extends class `A`, then a class `C`, which extends class `B`, will also inherit from class `A` via class `B`.
  - An object of the `SpotLightBulb` class *is-a* object of the class `Light`.
- The *is-a* relationship does not hold between peer classes: an object of the `LightBulb` class is not an object of the class `TubeLight`, and vice versa.
- *Litmus test for using inheritance*: if `B` *is* an `A`, then only let `B` inherit from `A`.

  i.e. do not use inheritance unless all inherited behavior makes sense.

Aggregation

- A major mechanism for code reuse mechanism is *aggregation*.
- Aggregation defines the relationship *has-a* (a.k.a. *whole–part* relationship) between an instance of a class and its constituents (a.k.a. *parts)*.
- In Java, an aggregate object cannot contain other objects.
  - It can only have *references* to its constituent objects.
- The *has-a* relationship defines an *aggregation hierarchy.*
- In this simple form of aggregation, constituent objects can be shared between objects, and their *lifetimes* are independent of the lifetime of the aggregate object.

Illustrating Inheritance

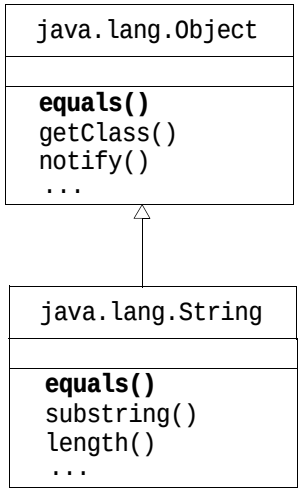


Figure 2 Inheritance Relationship between String and Object classes

Example 2 Illustrating Inheritance

```
// String class is a subclass of Object class
class Client {
    public static void main(String args[]) {

        String stringRef = new String("Java");                  // (1)

        System.out.println("(2): " + stringRef.getClass());     // (2)
        System.out.println("(3): " + stringRef.length());       // (3)
        Object objRef = stringRef;                              // (4)
   //   System.out.println("(5): " + objRef.length());          // (5) Not OK.
        System.out.println("(6): " + objRef.equals("Java"));    // (6)
        System.out.println("(7): " + objRef.getClass());        // (7)

        stringRef = (String) objRef;                            // (8)
        System.out.println("(9): " + stringRef.equals("C++"));  // (9)
    }
}
```

Output from the program:

```
(2): class java.lang.String
(3): 4
(6): true
(7): class java.lang.String
(9): false
```


Inheriting from the Superclass

- The subclass `String` inherits the method `getClass()` from the superclass `Object` - this is immaterial for a client of class `String`

```
System.out.println("(2): " + stringRef.getClass());     // (2)
```

Extending the Superclass

- The subclass `String` defines the method `length()`, which is not in the superclass `Object`, thereby extending the superclass.

```
System.out.println("(3): " + stringRef.length());       // (3)
```

Upcasting

- A subclass reference can be assigned to a superclass reference, because a subclass object can be used where a superclass object can be used.

```
Object objRef = stringRef;                              // (4) creates aliases
```

- Methods exclusive to the `String` subclass cannot be invoked via the superclass reference:

```
System.out.println("(5): " + objRef.length());          // (5) Not OK.
```

Method Overriding

- The `equals()` method is redefined in the `String` class with the same signature (i.e. method name and parameters) and the same return type.

```
System.out.println("(6): " + objRef.equals("Java"));    // (6)
```

- The compiler can check that the `Object` class does define a method called `equals()`.

Polymorphism and Dynamic Method Binding

- The ability of a superclass reference to denote objects of its own class and its subclasses at runtime is called *polymorphism*.
- The method invoked is dependent on the *actual (type of) object* denoted by the reference at runtime.
- The actual method is determined by *dynamic method lookup*.

```
System.out.println("(6): " + objRef.equals("Java"));    // (6)
System.out.println("(7): " + objRef.getClass());        // (7)
```

- At (6), dynamic method lookup results in the `equals()` method from the `String` class being executed, and not the one in the `Object` class.

- At (7), dynamic method lookup determines that the method `getClass()` inherited from the `Object` class to be executed - leading to a "search" up the inheritance hierarchy.

Downcasting

- Casting the value of a superclass reference to a subclass type is called *downcasting*, and requires explicit casting.

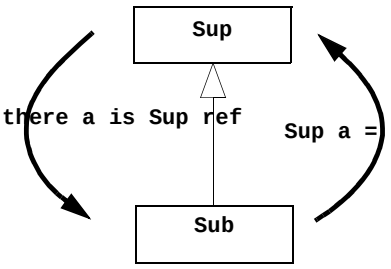
```
stringRef = (String) objRef;                          // (8)
System.out.println("(9): " + stringRef.equals("C++"));  // (9)
```

- *The cast can be invalid at runtime!*
  - A `ClassCastException` would be thrown at runtime.
  - Use the `instanceof` operator to determine the runtime type of an object before any cast is applied.

```
if (objRef instanceof String) {
    stringRef = (String) objRef;
    System.out.println("(9): " + stringRef.length);
}
```

Sub is Sup

```
if (a instanceof Sub) // there a is Sup ref
     Sub b = (Sub) a;
```




```
Sup a = b; // there b is Sub ref
```


Method Overriding

- A subclass may *override* non-static methods (non-private and non-final) inherited from the superclass.
- When the method is invoked on an object of the subclass, it is the new method definition in the subclass that is executed.
- The new method definition in the subclass must have the same *method signature* (i.e. method name and parameters) and the same return type.
  - The new method definition, in addition, cannot "narrow" the accessibility of the method, but it can "widen" it.
  - The new method definition in the subclass can only specify all or a subset of the exception classes (including their subclasses) specified in the `throws` clause of the overridden method in the superclass.
- A subclass can also use the keyword `super` to invoke the overridden method in the superclass.
- Any `final`, `static` and `private` methods in a class *cannot* be overridden but a subclass can redefine such methods although that would not be a good idea.

Example 3 Overriding and Overloading Methods and Shadowing Variables

```
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

    protected String billType = "Small bill";              // (1)

    protected double getBill(int noOfHours)
              throws InvalidHoursException {               // (2)
        double smallAmount = 10.0,
               smallBill = smallAmount * noOfHours;
        System.out.println(billType + ": " + smallBill);
        return smallBill;
    }
}

class TubeLight extends Light {

    public String billType = "Large bill";                 // (3) Shadowing.

    public double getBill(final int noOfHours)
           throws ZeroHoursException {                     // (4) Overriding.
        double largeAmount = 100.0,
               largeBill = largeAmount * noOfHours;
        System.out.println(billType + ": " + largeBill);
        return largeBill;
    }
```



```
    public double getBill() {                              // (5)
        System.out.println("No bill");
        return 0.0;
    }
}

public class Client {
    public static void main(String args[])
            throws InvalidHoursException {                 // (6)

        TubeLight tubeLightRef = new TubeLight();          // (7)
        Light lightRef1 = tubeLightRef;                    // (8)
        Light lightRef2 = new Light();                     // (9)

        // Invoke overridden methods
        tubeLightRef.getBill(5);                           // (10)
        lightRef1.getBill(5);                              // (11)
        lightRef2.getBill(5);                              // (12)

        // Access shadowed variables
        System.out.println(tubeLightRef.billType);         // (13)
        System.out.println(lightRef1.billType);            // (14)
        System.out.println(lightRef2.billType);            // (15)

        // Invoke overloaded method
        tubeLightRef.getBill();                            // (16)
    }
}
```

Output from the program:

```
Large bill: 500.0
Large bill: 500.0
Small bill: 50.0
Large bill
Small bill
Small bill
No bill
```


Variable Shadowing

- A subclass cannot override variable members of the superclass, but it can *shadow* them.
- A subclass method can use the keyword `super` to access inherited members, including shadowed variables.
- When a method is invoked on an object using a reference, it is the *class of the current object* denoted by the reference, not the type of the reference, that determines which method implementation will be executed.
- When a variable of an object is accessed using a reference, it is the *type of the reference*, not the class of the current object denoted by the reference, that determines which variable will actually be accessed.

Overriding vs. Overloading

- Method overriding requires the same *method signature* (name and parameters) and the same return type, and that the original method is inherited from its superclass.
- Overloading requires different method signatures, but the method name should be the same.
  - To overload methods, the parameters must differ in type or number.
  - The return type is not a part of the signature, changing it is not enough to overload methods.
- A method can be overloaded in the class it is defined in, or in a subclass of its class.
- Invoking an overridden method in the superclass from a subclass requires special syntax (for example, the keyword `super`).

Object Reference `super`

- The `this` reference is passed as an implicit parameter when an instance method is invoked.
  - It denotes the object on which the method is called.
- The keyword `super` can be used in the body of an instance method in a subclass to access variables and invoke methods inherited from the superclass.
  - The keyword `super` provides a reference to the current object as an instance of its superclass.
- The `super.super.X` construct is invalid.

Example 4 Using `super` *Keyword*

```
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

    protected String billType = "Small bill";              // (1)

    protected double getBill(int noOfHours)
              throws InvalidHoursException {               // (2)
        double smallAmount = 10.0,
               smallBill = smallAmount * noOfHours;
        System.out.println(billType + ": " + smallBill);
        return smallBill;
    }

    public void banner() {                                 // (3)
        System.out.println("Let there be light!");
    }
}
```



```
class TubeLight extends Light {

    public String billType = "Large bill";                 // (4) Shadowing.

    public double getBill(final int noOfHours)
           throws ZeroHoursException {                     // (5) Overriding.
        double largeAmount = 100.0,
               largeBill = largeAmount * noOfHours;
        System.out.println(billType + ": " + largeBill);
        return largeBill;
    }

    public double getBill() {                              // (6)
        System.out.println("No bill");
        return 0.0;
    }
}

class NeonLight extends TubeLight {
    // ...
    public void demonstrate()
            throws InvalidHoursException {                 // (7)
        super.banner();                                    // (8)
        super.getBill(20);                                 // (9)
        super.getBill();                                   // (10)
        System.out.println(super.billType);                // (11)
        ((Light) this).getBill(20);                        // (12)
        System.out.println(((Light) this).billType);       // (13)
    }
}
```

```
public class Client {
    public static void main(String args[])
                      throws InvalidHoursException {
        NeonLight neonRef = new NeonLight();
        neonRef.demonstrate();
    }
}
```

Output from the program:

```
Let there be light!
Large bill: 2000.0
No bill
Large bill
Large bill: 2000.0
Small bill
```


Constructor Overloading

- Constructors cannot be inherited or overridden.
- They can be overloaded, but only in the same class.

```
class Light {

    // Instance Variables
    private int noOfWatts;          // wattage
    private boolean indicator;      // on or off
    private String location;        // placement

    // Constructors
    Light() {                                       // (1) Explicit default constructor
        noOfWatts = 0;
        indicator = false;
        location = "X";
        System.out.println("Returning from default constructor no. 1.");
    }
    Light(int watts, boolean onOffState) {                    // (2) Non-default
        noOfWatts = watts;
        indicator = onOffState;
        location = "X";
        System.out.println("Returning from non-default constructor no. 2.");
    }
```

```
    Light(int noOfWatts, boolean indicator, String location) {  // (3) Non-default
        this.noOfWatts = noOfWatts;
        this.indicator = indicator;
        this.location = new String(location);
        System.out.println("Returning from non-default constructor no. 3.");
    }
}

public class DemoConstructorCall {
    public static void main(String args[]) {                    // (4)
        System.out.println("Creating Light object no.1.");
        Light light1 = new Light();
        System.out.println("Creating Light object no.2.");
        Light light2 = new Light(250, true);
        System.out.println("Creating Light object no.3.");
        Light light3 = new Light(250, true, "attic");
    }
}
```

Output from the program:

```
Creating Light object no.1.
Returning from default constructor no. 1.
Creating Light object no.2.
Returning from non-default constructor no. 2.
Creating Light object no.3.
Returning from non-default constructor no. 3.
```


`this()` Constructor Call

- The `this()` construct can be regarded as being “locally overloaded”.
- The `this()` call invokes the constructor with the corresponding parameter list.
- *Local chaining* of constructors in the class when an instance of the class is created.
- Java specifies that when using the `this()` call, it must occur as the *first* statement in a constructor, and it can only be used in a constructor definition.
- Note the order in which the constructors are invoked in the example.

Example 5 `this()` *Constructor Call*

```
class Light {

    // Instance Variables
    private int       noOfWatts;
    private boolean   indicator;
    private String    location;

    // Constructors
    Light() {                              // (1) Explicit default constructor
        this(0, false);
        System.out.println("Returning from default constructor no. 1.");
    }
    Light(int watt, boolean ind) {         // (2) Non-default
        this(watt, ind, "X");
        System.out.println("Returning from non-default constructor no. 2.");
    }
    Light(int noOfWatts, boolean indicator, String location) {  // (3) Non-default
        this.noOfWatts = noOfWatts;
        this.indicator = indicator;
        this.location = new String(location);
        System.out.println("Returning from non-default constructor no. 3.");
    }
}
```



```
public class DemoThisCall {
    public static void main(String args[]) {                    // (4)
        System.out.println("Creating Light object no.1.");
        Light light1 = new Light();                              // (5)
        System.out.println("Creating Light object no.2.");
        Light light2 = new Light(250, true);                     // (6)
        System.out.println("Creating Light object no.3.");
        Light light3 = new Light(250, true, "attic");            // (7)
    }

}
```

Output from the program:

```
Creating Light object no.1.
Returning from non-default constructor no. 3.
Returning from non-default constructor no. 2.
Returning from default constructor no. 1.
Creating Light object no.2.
Returning from non-default constructor no. 3.
Returning from non-default constructor no. 2.
Creating Light object no.3.
Returning from non-default constructor no. 3.
```

`super()` Constructor Call

- The `super()` construct is used in a subclass constructor to invoke constructors in the *immediate* superclass.
- This allows the subclass to influence the initialization of its inherited state when an object of the subclass is created.
- A `super()` call in the constructor of a subclass will result in the execution of the relevant constructor from the superclass, based on the arguments passed.
- The `super()` call must occur as the *first* statement in a constructor, and it can only be used in a constructor definition.
  - This implies that `this()` and `super()` calls cannot both occur in the same constructor.

Example 6 `super()` *Constructor Call*

```
class Light {
    // Instance Variables
    private int       noOfWatts;
    private boolean   indicator;
    private String    location;

    // Constructors
    Light() {                               // (1) Explicit default constructor
        this(0, false);
        System.out.println(
            "Returning from default constructor no. 1 in class Light");
    }
    Light(int watt, boolean ind) {                              // (2) Non-default
        this(watt, ind, "X");
        System.out.println(
            "Returning from non-default constructor no. 2 in class Light");
    }
    Light(int noOfWatts, boolean indicator, String location) {  // (3) Non-default
        super();                                                // (4)
        this.noOfWatts = noOfWatts;
        this.indicator = indicator;
        this.location = new String(location);
        System.out.println(
            "Returning from non-default constructor no. 3 in class Light");
    }
}
```

```
class TubeLight extends Light {
    // Instance variables
    private int tubeLength;
    private int colorNo;

    TubeLight(int tubeLength, int colorNo) {                    // (5) Non-default
        this(tubeLength, colorNo, 100, true, "Unknown");
        System.out.println(
            "Returning from non-default constructor no. 1 in class TubeLight");
    }

    TubeLight(int tubeLength, int colorNo, int noOfWatts,
              boolean indicator, String location) {             // (6) Non-default
        super(noOfWatts, indicator, location);                  // (7)
        this.tubeLength = tubeLength;
        this.colorNo = colorNo;
        System.out.println(
            "Returning from non-default constructor no. 2 in class TubeLight");
    }
}
public class Chaining {
    public static void main(String args[]) {
        System.out.println("Creating a TubeLight object.");
        TubeLight tubeLightRef = new TubeLight(20, 5);          // (8)
    }
}
```


Output from the program:

```
Creating a TubeLight object.
Returning from non-default constructor no. 3 in class Light
Returning from non-default constructor no. 2 in class TubeLight
Returning from non-default constructor no. 1 in class TubeLight
```

(subclass–superclass) Constructor Chaining

- The `this()` construct is used to "chain" constructors in the *same* class, and the constructor at the end of such a chain can invoke a superclass constructor using the `super()` construct.
- The `super()` construct leads to chaining of subclass constructors to superclass constructors.
- This chaining behavior guarantees that all superclass constructors are called, starting with the constructor of the class being instantiated, all the way up to the root of the inheritance hierarchy, which is always the `Object` class.
- Note that the body of the constructors is executed in the reverse order to the call order, as `super()` can only occur as the first statement in a constructor.

Default `super()` Call

- If a constructor does not have either a `this()` or a `super()` call as its first statement, then a `super()` call to the default constructor in the superclass is inserted. The code

```
class A {
    public A() {}
    // ...
}
class B extends A {
    // no constructors
    // ...
}
```

is equivalent to

```
class A {
    public A() { super(); }      // (1)
    // ...
}
class B extends A {
    public B() { super(); }      // (2)
    // ...
}
```

- If a class only defines non-default constructors (i.e. only constructors with parameters), then its subclasses cannot rely on the implicit behavior of a `super()` call being inserted.
  - This will be flagged as a compile time error.
  - The subclasses must then explicitly call a superclass constructor, using the `super()` construct with the right arguments.

```
class NeonLight extends TubeLight {
    // Instance Variable
    String sign;

    NeonLight() {                              // (1)
        super(10, 2, 100, true, "Roof-top");   // (2) Cannot be commented out.
        sign = "All will be revealed!";
    }
    // ...
}
```

- Subclasses without any declared constructors will fail to compile if the superclass does not have a default constructor and provides only non-default constructors.

Summary of `super()` Call Usage

Superclass	No constructors or only the default constructor	Only non-default constructors	Both default and non-default constructors
Subclass	Default super() call or explicit default super() call	Explicit non-default super() call	Default super() call or explicit default super() call or explicit non-default super() call

Interfaces

- Java provides *interfaces* which allow new type names to be introduced and used polymorphically, and also permit *multiple interface inheritance*.
- Interfaces support *programming by contract*.

Defining Interfaces

- An interface defines a *contract* by specifying prototypes of methods, and not their implementation.

```
<interface header> {
    <interface body>
}
```

- An interface is abstract by definition and therefore cannot be instantiated. It should also not be declared `abstract`.
- Reference variables of the interface type can be declared.

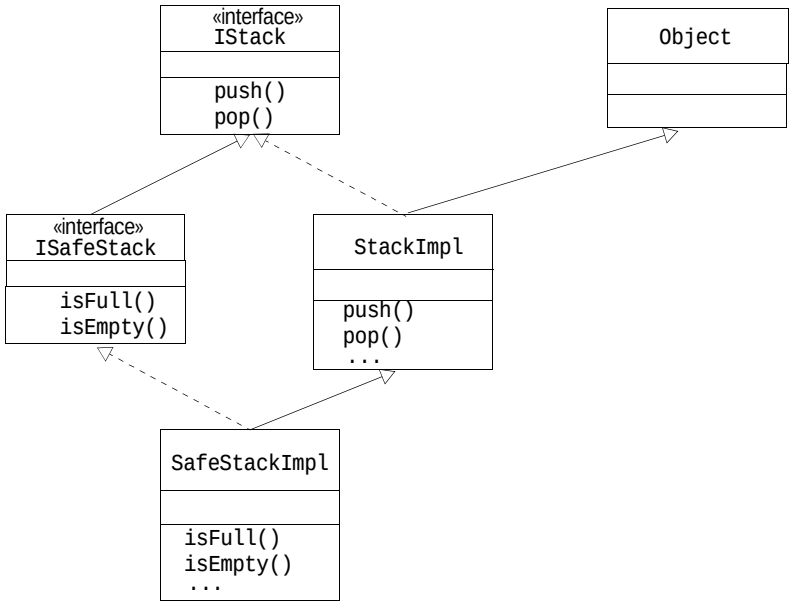


Figure 3 Inheritance Relations

Example 7 Interfaces

```
interface IStack {                                          // (1)
    void push(Object item);
    Object pop();
}

class StackImpl implements IStack {                         // (2)
    protected Object[] stackArray;
    protected int tos;

    public StackImpl(int capacity) {
        stackArray = new Object[capacity];
        tos = -1;
    }

    public void push(Object item)                           // (3)
        { stackArray[++tos] = item; }

    public Object pop() {                                   // (4)
        Object objRef = stackArray[tos];
        stackArray[tos] = null;
        tos--;
        return objRef;
    }

    public Object peek() { return stackArray[tos]; }
}
```

```
interface ISafeStack extends IStack {                              // (5)
    boolean isEmpty();
    boolean isFull();
}

class SafeStackImpl extends StackImpl implements ISafeStack {     // (6)

    public SafeStackImpl(int capacity) { super(capacity); }
    public boolean isEmpty() { return tos < 0; }                   // (7)
    public boolean isFull() { return tos >= stackArray.length; }  // (8)
}

public class StackUser {

    public static void main(String args[]) {                       // (9)
        SafeStackImpl safeStackRef = new SafeStackImpl(10);
        StackImpl stackRef = safeStackRef;
        ISafeStack isafeStackRef = safeStackRef;
        IStack istackRef = safeStackRef;
        Object objRef = safeStackRef;

        safeStackRef.push("Dollars");                              // (10)
        stackRef.push("Kroner");
        System.out.println(isafeStackRef.pop());
        System.out.println(istackRef.pop());
        System.out.println(objRef.getClass());
    }
}
```


Output from the program:

```
Kroner
Dollars
class SafeStackImpl
```

Implementing Interfaces

- Any class can elect to implement, wholly or partially, zero or more interfaces.
- Classes implementing interfaces thus introduce multiple interface inheritance into their linear implementation inheritance hierarchy.
- A class specifies the interfaces it implements as a comma-separated list using the `implements` clause in the class header.
- The interface methods will all have `public` accessibility when implemented in the class (or its subclasses).
- A class can choose to implement only some of the methods of its interfaces, i.e. give a partial implementation of its interfaces.
  - The class must then be declared as `abstract`.
- Note that interface methods cannot be declared `static`, because they comprise the contract fulfilled by the *objects* of the class implementing the interface and are therefore instance methods.

Extending Interfaces

- An interface can extend other interfaces, using the `extends` clause.
- Unlike extending classes, an interface can *extend* several interfaces.
- Multiple inheritance of interfaces can result in an inheritance hierarchy which has multiple roots designated by different interfaces.
- Note that there are three different inheritance relations at work when defining inheritance between classes and interfaces:
  - Linear implementation inheritance hierarchy between classes: a class extends another class.
  - Multiple inheritance hierarchy between interfaces: an interface extends other interfaces.
  - Multiple interface inheritance hierarchy between interfaces and classes: a class implements interfaces.
- There is only one single *implementation* inheritance into a class, which avoids many problems associated with general mutiple inheritance.

Supertypes

- Interfaces define new types.
- Although interfaces cannot be instantiated, variables of an interface type can be declared.
- If a class implements an interface, then references to objects of this class and its subclasses can be assigned to a variable of this interface type.
- The interfaces that a class implements and the classes it extends, directly or indirectly, are called its *supertypes*.
  - A supertype is thus a reference type.
- Interfaces with empty bodies are often used as markers to "tag" classes as having a certain property or behavior (`java.io.Serializable`).
- Note that a class inherits *only one* implementation of a method, regardless of how many supertypes it has.

Constants in Interfaces

- An interface can also define constants.
- Such constants are considered to be `public`, `static` and `final`.
- An interface constant can be accessed by any client (a class or interface) using its fully qualified name, regardless of whether the client extends or implements its interface.
- A class that implements this interface or an interface that extends this interface, can also access such constants directly without using the dot (.) notation.
- Extending an interface which has constants is analogous to extending a class having static variables.
  - In particular, these constants can be shadowed by the subinterfaces.
- In the case of multiple inheritance, any name conflicts can be resolved using fully qualified names for the constants involved.
  - The compiler will flag such conflicts.

Example 8 Variables in Interfaces

```
interface Constants {
    double PI = 3.14;
    String AREA_UNITS = " sq.cm.";
    String LENGTH_UNITS = " cm.";
}

public class Client implements Constants {
    public static void main(String args[]) {
        double radius = 1.5;
        System.out.println("Area of circle is " + (PI*radius*radius) +
                           AREA_UNITS);            // (1) Direct access.
        System.out.println("Circumference of circle is " + (2*PI*radius) +
                           Constants.LENGTH_UNITS); // (2) Fully qualified name.
    }
}
```

Output from the program:

```
Area of circle is 7.0649999999999995 sq.cm.
Circumference of circle is 9.42 cm.
```


Types in Java

	Corresponding Types:
Primitive data values	Primitive datatypes.
Reference values	Class, interface or array type (called *reference types*).
Objects	Class or array type.

- Only primitive data and reference values can be stored in variables.
- Arrays are objects in Java.
- Array types (`boolean[]`, `Object[]`, `StackImpl[]`) implicitly augment the inheritance hierarchy.
- All array types implicitly extend the `Object` class
- Note the difference between arrays of primitive datatypes and class types.
  - Arrays of reference types also extend the array type `Object[]`.
- Variables of array reference types can be declared, and arrays of reference types can be instantiated.
- An array reference exhibits the same polymorphic behavior as any other reference, subject to its location in the extended inheritance hierarchy.

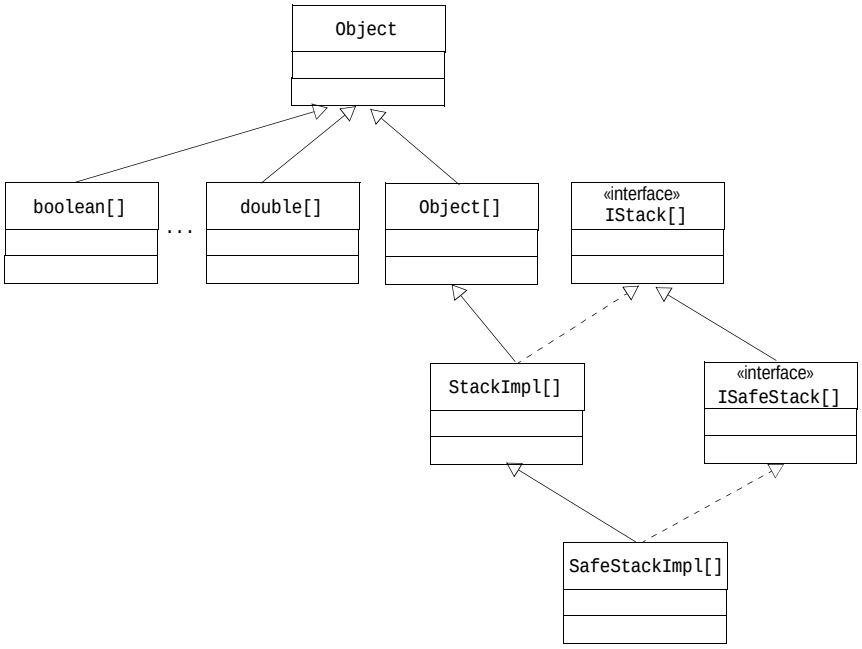


Figure 4 Array Types in Inheritance Hierarchy

Assigning, Passing and Casting References

- Reference values, like primitive values, can be assigned, cast and passed as arguments.
- For values of the primitive datatypes and reference types, conversions occur during:
  - Assignment
  - Parameter passing
  - Explicit casting
- The rule of thumb for the primitive datatypes is that widening conversions are permitted, but narrowing conversions require an explicit cast.
- The rule of thumb for reference values is that conversions up the inheritance hierarchy are permitted (called *upcasting*), but conversions down the hierarchy require explicit casting (called *downcasting*).
- *The parameter passing conversion rules are useful in creating generic data types which can handle objects of arbitrary types.*

Example 9 Assigning and Passing Reference Values

```
interface IStack { /* See Example 7 for definition */ }
class StackImpl implements IStack { /* See Example 7 for definition */ }
interface ISafeStack extends IStack { /* See Example 7 for definition */ }
class SafeStackImpl extends StackImpl implements ISafeStack {
    /* See Example 7 for definition */
}

public class ReferenceConversion {

    public static void main(String args[]) {

        Object objRef;
        StackImpl stackRef;
        SafeStackImpl safeStackRef = new SafeStackImpl(10);
        IStack iStackRef;
        ISafeStack iSafeStackRef;

        // SourceType is a class type
        objRef = safeStackRef;       // (1) Always possible
        stackRef = safeStackRef;     // (2) Subclass to superclass assignment
        iStackRef = stackRef;        // (3) StackImpl implements IStack
        iSafeStackRef = safeStackRef;// (4) SafeStackImpl implements ISafeStack

        // SourceType is an interface type
        objRef = iStackRef;          // (5) Always possible
        iStackRef = iSafeStackRef;   // (6) Sub- to super-interface assignment
```



```
        // SourceType is an array type.
        Object[] objArray = new Object[3];
        StackImpl[] stackArray = new StackImpl[3];
        SafeStackImpl[] safeStackArray = new SafeStackImpl[5];
        ISafeStack[] iSafeStackArray = new SafeStackImpl[5];
        int[] intArray = new int[10];

        objRef = objArray;           // (7) Always possible
        objRef = stackArray;         // (8) Always possible
        objArray = stackArray;       // (9) Always possible
        objArray = iSafeStackArray;  // (10) Always possible
        objRef = intArray;           // (11) Always possible
    //  objArray = intArray;         // (12) Compile time error
        stackArray = safeStackArray; // (13) Subclass array to superclass array
        iSafeStackArray =
                safeStackArray;      // (14) SafeStackImpl implements ISafeStack

        // Parameter Conversion
        System.out.println("First call:");
        sendParams(stackRef, safeStackRef, iStackRef,
                   safeStackArray,iSafeStackArray);                    // (15)
    //  Call Signature: sendParams(StackImpl, SafeStackImpl, IStack,
    //                             SafeStackImpl[], ISafeStack[]);

        System.out.println("Second call:");
        sendParams(iSafeStackArray, stackRef, iSafeStackRef,
                   stackArray, safeStackArray);                        // (16)
    //  Call Signature: sendParams(ISafeStack[], StackImpl, ISafeStack,
    //                             StackImpl[], SafeStackImpl[]);
    }
```

```
    public static void sendParams(Object objRefParam, StackImpl stackRefParam,
            IStack iStackRefParam, StackImpl[] stackArrayParam,
            IStack[] iStackArrayParam) {                              // (17)
    //  Signature: sendParams(Object, StackImpl, IStack, StackImpl[], IStack[])
    //  Print class name of object denoted by the reference at runtime.
        System.out.println(objRefParam.getClass());
        System.out.println(stackRefParam.getClass());
        System.out.println(iStackRefParam.getClass());
        System.out.println(stackArrayParam.getClass());
        System.out.println(iStackArrayParam.getClass());
    }
}
```

Output from the program:

```
First call:
class SafeStackImpl
class SafeStackImpl
class SafeStackImpl
class [LSafeStackImpl;
class [LSafeStackImpl;
Second call:
class [LSafeStackImpl;
class SafeStackImpl
class SafeStackImpl
class [LSafeStackImpl;
class [LSafeStackImpl;
```


Reference Casting and `instanceof` Operator

- The expression to cast <*reference*> of <*source type*> to <*destination type*> has the following syntax:

  (<*destination type*>) <*reference*>

- The binary `instanceof` operator has the following syntax:

  <*reference*> `instanceof` <*destination type*>

- The `instanceof` operator (note that the keyword is composed of only lowercase letters) returns the value `true` if the left-hand operand (any reference) can be *cast* to the right-hand operand (a class, interface or array type).
- A compile time check determines whether a reference of <*source type*> and a reference of <*destination type*> can denote objects of a class (or its subclasses) where this class is a common subtype of both <*source type*> and <*destination type*> in the inheritance hierarchy.
- At runtime, it is the actual object denoted by the reference that is compared with the type specified on the right-hand side.
- Typical usage of the `instanceof` operator is to determine what object a reference is denoting.

Example 10 `instanceof` *and cast Operator*

```
class Light { /* ... */ }
class LightBulb extends Light { /* ... */ }
class SpotLightBulb extends LightBulb { /* ... */ }
class TubeLight extends Light { /* ... */ }
class NeonLight extends TubeLight { /* ... */ }

public class WhoAmI {
    public static void main(String args[]) {
        boolean result1, result2, result3, result4, result5;
        Light light1 = new LightBulb();           // (1)
    //  String str = (String) light1;             // (2) Compile time error.
    //  result1 = light1 instanceof String;       // (3) Compile time error.

        result2 = light1 instanceof TubeLight;    // (4) false. Peer class.
    //  TubeLight tubeLight1 = (TubeLight) light1;// (5) ClassCastException.

        result3 = light1 instanceof SpotLightBulb;// (6) false: Superclass
    //  SpotLightBulb spotRef = (SpotLightBulb) light1;// (7) ClassCastException

        light1 = new NeonLight();                      // (8)
        if (light1 instanceof TubeLight) {             // (9) true
            TubeLight tubeLight2 = (TubeLight) light1; // (10) OK
            // Can now use tubeLight2 to access object of class NeonLight.
        }
    }
}
```


Converting References of Class and Interface Types

- Upcasting: References of interface type can be declared, and these can denote objects of classes that implement this interface.
- Downcasting: converting a reference of interface type to the type of the class implementing the interface requires explicit casting.

```
IStack istackOne = new StackImpl(5);               // Upcasting
StackImpl stackTwo = (StackImpl) istackOne;        // Downcasting

Object obj1 = istackOne.pop();       // OK. Method in IStack interface.
Object obj2 = istackOne.peek();      // Not OK. Method not in IStack interface.
```

Polymorphism and Dynamic Method Lookup

- Which object a reference will actually denote during runtime cannot always be determined at compile time.
- Polymorphism allows a reference to denote different objects in the inheritance hierarchy at different times during execution.
  - Such a reference is a supertype reference.
- When a method is invoked using a reference, the method definition which actually gets executed is determined both by *the class of the object* denoted by the reference at runtime and *the method signature.*
- Dynamic method lookup is the process of determining which method definition a method signature denotes during runtime, based on the class of the object.
- Polymorphism and dynamic method lookup form a powerful programming paradigm which simplifies client definitions, encourages object decoupling and supports dynamically changing relationships between objects at runtime.

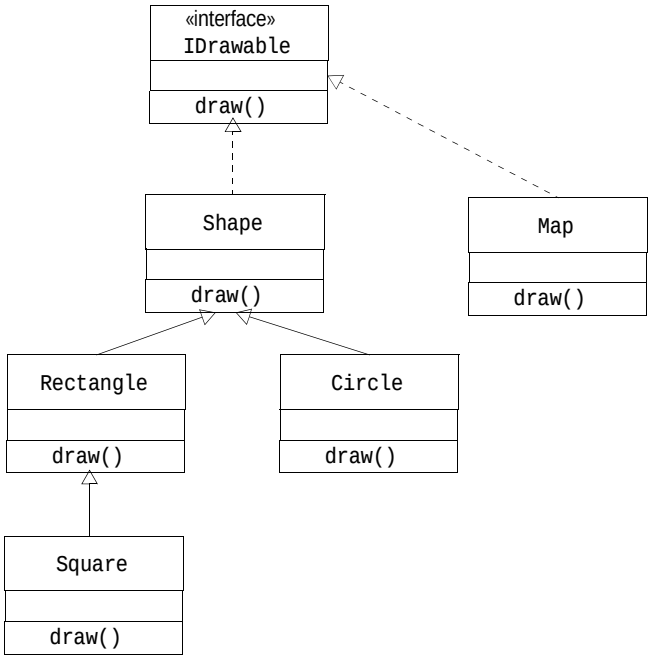


Figure 5 Polymorphic Methods

Example 11 Polymorphism and Dynamic Method Lookup

```
interface IDrawable {
    void draw();
}

class Shape implements IDrawable {
    public void draw() { System.out.println("Drawing a Shape."); }
}

class Circle extends Shape {
    public void draw() { System.out.println("Drawing a Circle."); }
}

class Rectangle extends Shape {
    public void draw() { System.out.println("Drawing a Rectangle."); }
}

class Square extends Rectangle {
    public void draw() { System.out.println("Drawing a Square."); }
}

class Map implements IDrawable {
    public void draw() { System.out.println("Drawing a Map."); }
}
```



```
public class PolymorphRefs {
    public static void main(String args[]) {
        Shape[] shapes = {new Circle(), new Rectangle(), new Square()};   // (1)
        IDrawable[] drawables = {new Shape(), new Rectangle(), new Map()};// (2)

        System.out.println("Draw shapes:");
        for (int i = 0; i < shapes.length; i++)                           // (3)
            shapes[i].draw();

        System.out.println("Draw drawables:");
        for (int i = 0; i < drawables.length; i++)                        // (4)
            drawables[i].draw();
    }
}
```

Output from the program:

```
Draw shapes:
Drawing a Circle.
Drawing a Rectangle.
Drawing a Square.
Draw drawables:
Drawing a Shape.
Drawing a Rectangle.
Drawing a Map.
```

Choosing between Inheritance and Aggregation

- Choosing between inheritance and aggregation to model relationships can be a crucial design decision.
- A good design strategy advocates that inheritance should be used only if the relationship *is-a* is unequivocally maintained throughout the lifetime of the objects involved, otherwise aggregation is the best choice.
- A *role* is often confused with an *is-a* relationship.
  - Changing roles would involve a new object to represent the new role every time this happened.
- Code reuse is also best achieved by aggregation when there is no *is-a* relationship.
- Aggregation with *method delegating* can result in robust abstractions.
- Both inheritance and aggregation promote encapsulation of *implementation*, as changes to the implementation are localized to the class.
- Changing the *contract* of a superclass can have consequences for the subclasses (called the *ripple effect*) and also for clients who are dependent on a particular behavior of the subclasses.

Achieving Polymorphism

- Polymorphism is achieved through inheritance and interface implementation.
- Code relying on polymorphic behavior will still work without any change if new subclasses or new classes implementing the interface are added.
- If no obvious *is-a* relationship is present, then polymorphism is best achieved by using *aggregation with interface implementation*.

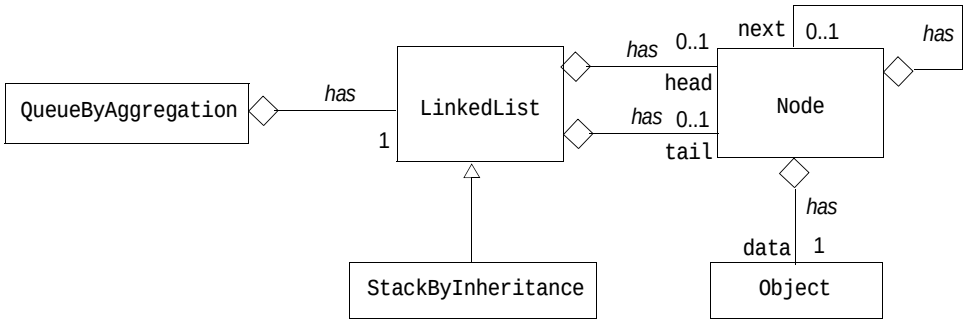


Figure 6 Inheritance and Aggregation

Example 12 Inheritance and Aggregation

```
class Node {                                                    // (1)
    private Object data;    // Data
    private Node next;      // Next node

    // Constructor for initializing data and reference to the next node.
    public Node(Object obj, Node link) {
        data = obj;
        next = link;
    }

    // Accessor methods
    public void setData(Object obj) { data = obj; }
    public Object getData() { return data; }
    public void setNext(Node node) { next = node; }
    public Node getNext() { return next; }
}

class LinkedList {                                              // (2)

    protected Node head = null;
    protected Node tail = null;
    // Modifier methods
    public void insertInFront(Object dataObj) {
        if (isEmpty()) head = tail = new Node(dataObj, null);
        else head = new Node(dataObj, head);
    }
```

```
    public void insertAtBack(Object dataObj) {
        if (isEmpty())
            head = tail = new Node(dataObj, null);
        else {
            tail.setNext(new Node(dataObj, null));
            tail = tail.getNext();
        }
    }
    public Object deleteFromFront() {
        if (isEmpty()) return null;
        Node removed = head;
        if (head == tail) head = tail = null;
        else head = head.getNext();
        return removed.getData();
    }
    // Selector method
    public boolean isEmpty() { return head == null; }
}

class QueueByAggregation {                                    // (3)
    private LinkedList qList;
    // Constructor
    public QueueByAggregation() {
        qList = new LinkedList();
    }
    // Methods
    public void enqueue(Object item) { qList.insertAtBack(item); }
```



```
    public Object dequeue() {
        if (empty()) return null;
        else return qList.deleteFromFront();
    }
    public Object peek() {
        Object obj = dequeue();
        if (obj != null) qList.insertInFront(obj);
        return obj;
    }
    public boolean empty() { return qList.isEmpty(); }
}

class StackByInheritance extends LinkedList {                // (4)
    public void push(Object item) { insertInFront(item); }
    public Object pop() {
        if (empty()) return null;
        else return deleteFromFront();
    }
    public Object peek() {
        return (isEmpty() ? null : head.getData());
    }
    public boolean empty() { return isEmpty(); }
}
```

```
public class Client {                                          // (5)
    public static void main(String args[]) {
        String string1 = "Queues are boring to stand in!";
        int length1 = string1.length();
        QueueByAggregation queue = new QueueByAggregation();
        for (int i = 0; i<length1; i++)
            queue.enqueue(new Character(string1.charAt(i)));
        while (!queue.empty())
            System.out.print((Character) queue.dequeue());
        System.out.println();

        String string2 = "!no tis ot nuf era skcatS";
        int length2 = string2.length();
        StackByInheritance stack = new StackByInheritance();
        for (int i = 0; i<length2; i++)
            stack.push(new Character(string2.charAt(i)));
        stack.insertAtBack(new Character('!'));                // (6)
        while (!stack.empty())
            System.out.print((Character) stack.pop());
        System.out.println();

    }
}
```

Output from the program:

```
Queues are boring to stand in!
Stacks are fun to sit on!
```


Encapsulation

- Encapsulation helps to make clear the distinction between an object's contract and implementation.
- Encapsulation has major consequences for program development.
  - Results in programs that are "black boxes".
  - Implementation of an object can change without implications for the clients.
  - Reduces dependency between program modules (hence complexity), as the internals of an object are hidden from the clients, who cannot influence its implementation.
  - Encourages code-reuse.

Encapsulation Levels