Object-Oriented Programming

Single Implementation Inheritance

- Inheritance is one of the fundamental mechanisms for code reuse in OOP. It allows new classes to be derived from an existing class.
- The new class (also called **subclass**, **subtype**, **derived class**, **child class**) can inherit members from the old class (also called **superclass**, **supertype**, **base class**, **parent class**).
- The subclass can add new behavior and properties and, under certain circumstances, modify its inherited behavior.

Single Implementation Inheritance

In Java, implementation inheritance is achieved by extending classes (i.e., adding new fields and methods) and modifying inherited members

Single Implementation Inheritance

- Inheritance of members is closely tied to their declared accessibility. If a superclass member is accessible by its simple name in the subclass (without the use of any extra syntax like super), that member is considered inherited.
- This means that private, overridden, and hidden members of the superclass are not inherited
- Inheritance should not be confused with the *existence* of such members in the state of a subclass object

Single Implementation Inheritance

- The superclass is specified using the extends clause in the header of the subclass declaration.
- The subclass only specifies the additional new and modified members in its class body.
- The rest of its declaration is made up of its inherited members.

Single Implementation Inheritance

- If no extends clause is specified in the header of a class declaration, the class implicitly inherits from the java.lang.Object class.
- This implicit inheritance is assumed in the declaration of the Light class at (1) in Example

```
class Light {              // (1)
  // Instance fields:
  int noOfWatts;    // wattage
  private boolean indicator;  // on or off
  protected String  location; // placement
  // Static field:
  private static int counter;   // no. of Light objects created
  // Constructor:
  Light() {
    noOfWatts = 50;
    indicator = true;
    location  = "X";
    counter++;
  }
  // Instance methods:
  public void switchOn() { indicator = true; }
  public void switchOff(){ indicator = false; }
  public boolean isOn()  { return indicator; }
  private void  printLocation() {
    System.out.println("Location: " + location);
  }
  // Static methods:
  public static void writeCount() {
    System.out.println("Number of lights: " + counter);
  }
  //...
}
```

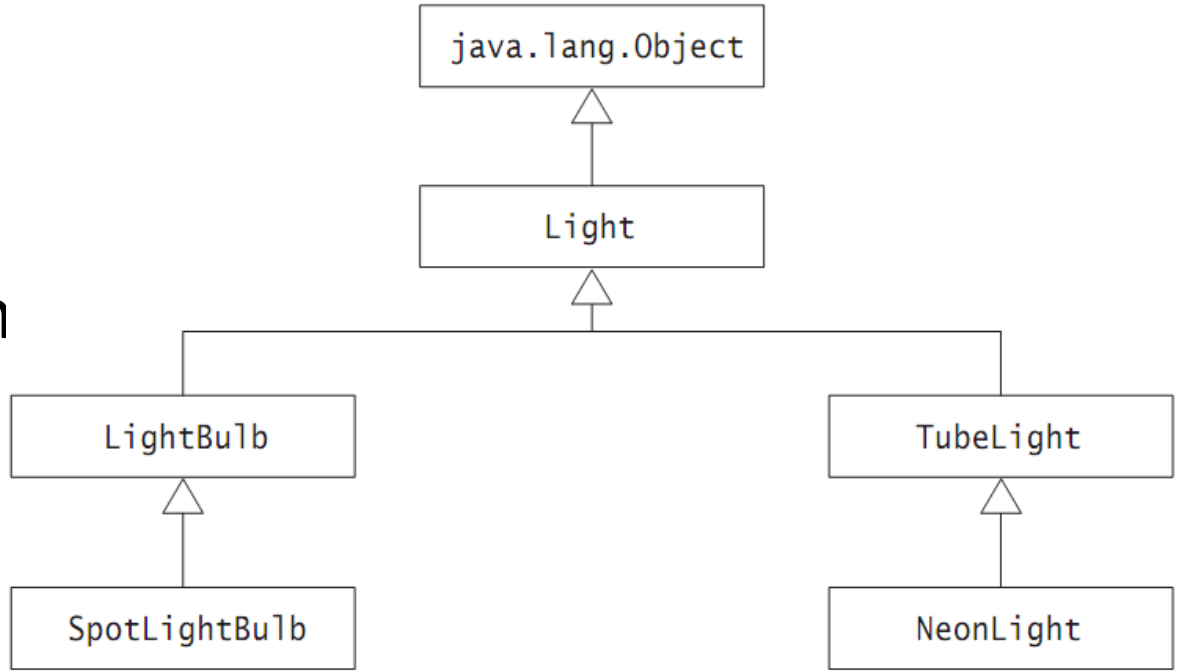
```
class TubeLight extends Light { // (2) extends
  // Instance fields:
  private int tubeLength = 54;
  private int colorNo    = 10;
  // Instance methods:
  public int getTubeLength() { return tubeLength; }
  public void printInfo() {
    System.out.println("Tube length: "  + getTubeLength());
    System.out.println("Color number: " + colorNo);
    System.out.println("Wattage: "      + noOfWatts);     // Inherited.
//  System.out.println("Indicator: "    + indicator);     // Not Inherited.
    System.out.println("Indicator: "    + isOn());        // Inherited.
    System.out.println("Location: "     + location);      // Inherited.
//  printLocation();                                      // Not Inherited.
//  System.out.println("Counter: "   + counter);          // Not Inherited.
    writeCount();                                         // Inherited.
  }
  // ...
}
//_____________________________________________________________________________
public class Utility {          // (3)
  public static void main(String[] args) {
    new TubeLight().printInfo();
  }
}
```

Inheritance Hierarchy

- In Java, a class can only extend one other class; i.e., it can only have one immediate superclass. This kind of inheritance is sometimes called **single** or **linear implementation** inheritance.
- The name is appropriate, as the subclass inherits the implementations of its superclass members. The inheritance relationship can be depicted as an **inheritance hierarchy** (also called **class hierarchy**).

Relationships: **is-a** and **has-a**

Inheritance defines the relationship is-a (also called the superclass–subclass relationship) between a superclass and its subclasses.



Relationships: **is-a** and **has-a**

- Whereas **inheritance** defines the relationship **is-a** between a superclass and its subclasses, **aggregation** defines the relationship **has-a** (also called the ***whole–part relationship***) between an instance of a class and its constituents (also called parts).
- Aggregation comprises the usage of objects. An instance of class Light **has** (or **uses**) the following parts:
  - a field to store its wattage (**noOfWatts**),
  - a field to store whether it is on or off (**indicator**), and
  - a String object to store its location (denoted by the field reference **location**)

The Supertype-Subtype Relationship

A class defines a reference type. Therefore the inheritance hierarchy can be regarded as a type hierarchy, embodying the supertype-subtype relationship between reference types.

The Supertype-Subtype Relationship

- In the context of Java, the supertype-subtype relationship implies that the reference value of a subtype object can be assigned to a supertype reference, because a subtype object can be substituted for a supertype object.
- This assignment involves a widening reference conversion as references are assigned up the inheritance hierarchy.

The Supertype-Subtype Relationship

Examlple:

```
Light light = new TubeLight();  // (1) widening reference conversion
```

We can now use the reference `light` to invoke those methods on the subtype object that are inherited from the supertype `Light`:

```
light.switchOn();               // (2)
```

Note that the compiler only knows about the declared type of the reference `light`, which is `Light`, and ensures that only methods from this type can be called using the reference `light`. However, at runtime, the reference `light` will refer to an object of the subtype `TubeLight` when the call to the method `switchOn()` is executed. It is the type of the object that the reference is referring to at runtime that determines which method is executed.

The Supertype-Subtype Relationship

One might be tempted to invoke methods exclusive to the TubeLight subtype via the supertype reference light:

```
light.getTubeLength();                    // (3) Not OK.
```

However, this will not work, as the compiler does not know what object the reference `light` is denoting. It only knows the declared type of the reference. As the declaration of the class `Light` does not have a method called `getTubeLength()`, this method call at (3) results in a compile-time error.

Overriding Methods

Instance Method Overriding

Under certain circumstances, a subclass may override instance methods that it would otherwise inherit from a superclass.

The overridden method in the superclass is not inherited by the subclass, and the new method in the subclass must abide by the following rules of method overriding:

Instance Method Overriding

- The new method definition must have the same method signature, i.e., the method name, and the types and the number of parameters, including their order, are the same as in the overridden method.
  - Whether parameters in the overriding method should be final is at the discretion of the subclass. A method's signature does not comprise the final modifier of parameters, only their types and order.
- The return type of the overriding method *can be a subtype* of the return type of the overridden method (called ***covariant*** return).
- The new method definition *cannot narrow* the accessibility of the method, but it *can widen* it.
- The new method definition can only throw all or none, or a subset of the checked exceptions (including their subclasses) that are specified in the throws clause of the overridden method in the superclass.

```
//Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}
class Light {
  protected String billType = "Small bill";  // (1) Instance field
  protected double getBill(int noOfHours)
                throws InvalidHoursException { // (2) Instance method
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double smallAmount = 10.0, smallBill = smallAmount * noOfHours;
    System.out.println(billType + ": " + smallBill);
    return smallBill;
  }
  public Light makeInstance() {      // (3) Instance method
    return new Light();
  }
  public static void printBillType() {    // (4) Static method
    System.out.println("Small bill");
  }
}
```

```
class TubeLight extends Light {
  public static String billType = "Large bill";  // (5) Hiding field at (1)
  @Override
  public double getBill(final int noOfHours)
      throws ZeroHoursException { // (6) Overriding instance method at (2)
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double largeAmount = 100.0, largeBill = largeAmount * noOfHours;
    System.out.println(billType + ": " + largeBill);
    return largeBill;
  }
  public double getBill() {                  // (7) Overloading method at (6)
    System.out.println("No bill");
    return 0.0;
  }
  @Override
  public TubeLight makeInstance() {// (8) Overriding instance method at (3)
    return new TubeLight();
  }
  public static void printBillType() {  // (9) Hiding static method at (4).
    System.out.println(billType);
  }
}
```

```
public class Client {
  public static void main(String[] args) throws InvalidHoursException { // (10)
    TubeLight tubeLight = new TubeLight();                               // (11)
    Light     light1    = tubeLight;                                  // (12) Aliases
    Light     light2    = new Light();                                // (13)
    System.out.println("Invoke overridden instance method:");
    tubeLight.getBill(5);   // (14) Invokes method at (6).
    light1.getBill(5);    // (15) Invokes method at (6).
    light2.getBill(5);    // (16) Invokes method at (2).
    System.out.println("Invoke overridden instance method with covariant return:");
    System.out.println(light2.makeInstance().getClass()); // (17) Invokes method at (3).
    System.out.println(tubeLight.makeInstance().getClass()); //(18)Invokes method at (8)
    System.out.println("Access hidden field:");
    System.out.println(tubeLight.billType);   // (19) Accesses field at (5).
    System.out.println(light1.billType);      // (20) Accesses field at (1).
    System.out.println(light2.billType);      // (21) Accesses field at (1).
    System.out.println("Invoke hidden static method:");
    tubeLight.printBillType();                // (22) Invokes method at (9).
    light1.printBillType();                   // (23) Invokes method at (4).
    light2.printBillType();                   // (24) Invokes method at (4).
    System.out.println("Invoke overloaded method:");
    tubeLight.getBill();                      // (25) Invokes method at (7).
  }
}
```

More facts about overriding

- A subclass must use the keyword super in order to invoke an overridden method in the superclass
- An instance method in a subclass cannot override a static method in the superclass.
- However, a static method in a subclass can hide a static method in the superclass
- A final method cannot be overridden
- The accessibility modifier private for a method means that the method is not accessible outside the class in which it is defined; therefore, a subclass cannot override it.

Overriding vs. Overloading

Comparison Criteria	Overriding	Overloading
Method name	Must be the same.	Must be the same.
Argument list	Must be the same.	Must be different.
Return type	Can be the same type or a covariant type.	Can be different.
`throws` clause	Must not throw new checked exceptions. Can narrow exceptions thrown.	Can be different.
Accessibility	Can make it less restrictive, but not more restrictive.	Can be different.
Declaration context	A method can only be overridden in a subclass.	A method can be overloaded in the same class or in a subclass.
Method call resolution	The *runtime type* of the reference, i.e., the type of the object referenced at *runtime*, determines which method is selected for execution.	At compile time, the *declared type* of the reference is used to determine which method will be executed at runtime.

Hiding Members - Field Hiding

- A subclass cannot override fields of the superclass, but it can hide them.
- If this is the case, the fields in the superclass cannot be accessed in the subclass by their simple names
- Code in the subclass can use the keyword super to access such members, including hidden fields.
- A client can use a reference of the superclass to access members that are hidden in the subclass
- If the hidden field is static, it can also be accessed by the superclass name

Hiding Members - Static Method Hiding

- A static method cannot override an inherited instance method, but it can hide a static method if the exact requirements for overriding instance methods are fulfilled
- The compiler will flag an error if the signatures are the same, but the other requirements regarding return type, throws clause, and accessibility are not met.
- If the signatures are different, the method name is overloaded, not hidden.
- A call to a static or final method is bound to a method implementation at compile time (private methods are implicitly final).

The Object Reference `super`

- The keyword super can be used in non-static code (e.g., in the body of an instance method), but only in a subclass, to access fields and invoke methods from the superclass
- The keyword super provides a reference to the current object as an instance of its superclass
- Typically used to invoke methods that are overridden and to access members that are hidden in the subclass
- Unlike the this keyword, the super keyword cannot be used as an ordinary reference (it cannot be assigned to other references or cast to other reference types)

```
class Light {
  protected String billType  = "Small bill";        // (1)
  protected double getBill(int noOfHours)
  throws InvalidHoursException {                    // (2)
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double smallAmount = 10.0, smallBill = smallAmount * noOfHours;
    System.out.println(billType + ": " + smallBill);
    return smallBill;
  }
  public static void printBillType() {              // (3)
    System.out.println("Small bill");
  }
  public void banner() {                            // (4)
    System.out.println("Let there be light!");
  }
}
```

```
class TubeLight extends Light {
  public static String billType = "Large bill";
  // ^^^ (5) Hiding static field at (1).
  @Override
  public double getBill(final int noOfHours) throws ZeroHoursException {
            // (6) Overriding instance method at (2).
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double largeAmount = 100.0, largeBill = largeAmount * noOfHours;
    System.out.println(billType + ": " + largeBill);
    return largeBill;
  }
  public static void printBillType() { // (7) Hiding static method at (3)
    System.out.println(billType);
  }
  public double getBill() {       // (8) Overloading method at (6).
    System.out.println("No bill");
    return 0.0;
  }
}
```

```
class NeonLight extends TubeLight {
  // ...
  public void demonstrate() throws InvalidHoursException {     // (9)
    super.banner();                          // (10) Invokes method at (4)
    super.getBill(20);                       // (11) Invokes method at (6)
    super.getBill();                         // (12) Invokes method at (8)
    ((Light) this).getBill(20);              // (13) Invokes method at (6)
    System.out.println(super.billType);  // (14) Accesses field at (5)
    System.out.println(((Light) this).billType); // (15)
                                                  //Accesses field at (1)
    super.printBillType();                   // (16) Invokes method at (7)
    ((Light) this).printBillType();          // (17) Invokes method at (3)
  }
}
```

Chaining Constructors Using **this()** and **super()**

- Constructors cannot be inherited or overridden.
- They can be overloaded, but only in the same class.
- Since a constructor always has the same name as the class, each parameter list must be different when defining more than one constructor for a class.
- The **`this()`** call invokes the local constructor with the corresponding parameter list
- Java requires that any **`this()`** call must occur as the first statement in a constructor.

Chaining Constructors Using **this()** and **super()**

- The **`super()`** construct is used in a subclass constructor to invoke a constructor in the immediate superclass.
- A super() call in the constructor of a subclass will result in the execution of the relevant constructor from the superclass, based on the signature of the call.

Chaining Constructors Using **this()** and **super()**

- A constructor in a subclass can access the class's inherited members by their simple names.
- The keyword super can also be used in a subclass constructor to access inherited members via its superclass.
- One might be tempted to use the super keyword in a constructor to specify initial values of inherited fields.
- However, the super() construct provides a better solution to initialize the inherited state.

Chaining Constructors Using **this()** and **super()**

- The **`super()`** construct has the same restrictions as the **`this()`** construct: if used, the **`super()`** call must occur as the first statement in a constructor, and it can only be used in a constructor declaration.
- This implies that **`this()`** and **`super()`** calls cannot both occur in the same constructor.

Interfaces

```
<accessibility modifier> interface <interface name>
     <extends interface clause> // Interface header
{ // Interface body
<constant declarations>
<abstract method declarations>
<nested class declarations>
<nested interface declarations>
}
```

Interfaces

The interface header can specify the following information:

- scope or accessibility modifier
- any interfaces it extends

The interface body can contain member declarations which comprise:

- constant declarations
- abstract method declarations
- nested class and interface declarations

Interfaces

- An interface does not provide any implementation and is, therefore, abstract by definition.
- This means that it cannot be instantiated.
- Declaring an interface abstract is superfluous and seldom done.
- Since interfaces are meant to be implemented by classes, interface members implicitly have public accessibility and the public modifier can be omitted.

Interfaces

- Interfaces with empty bodies can be used as markers to tag classes as having a certain property or behavior.
- Such interfaces are also called ability interfaces.
- Java APIs provide several examples of such marker interfaces:
  - `java.lang.Cloneable`
  - `java.io.Serializable`
  - `java.util.EventListener`

Abstract Method Declarations

- An interface defines a contract by specifying a set of abstract method declarations, but provides no implementations
- The methods in an interface are all implicitly **abstract** and **public** by virtue of their definition.
- Only the modifiers **abstract** and **public** are allowed, but these are invariably omitted.

```
<optional type parameter list> <return type>
        <method name> (<parameter list>)
                                <throws clause>;
```

Implementing Interfaces

- Any class can elect to implement, wholly or partially, zero or more interfaces.
- A class specifies the interfaces it implements as a comma-separated list of unique interface names in an implements clause in the class header.
- The interface methods must all have public accessibility when implemented in the class (or its subclasses).

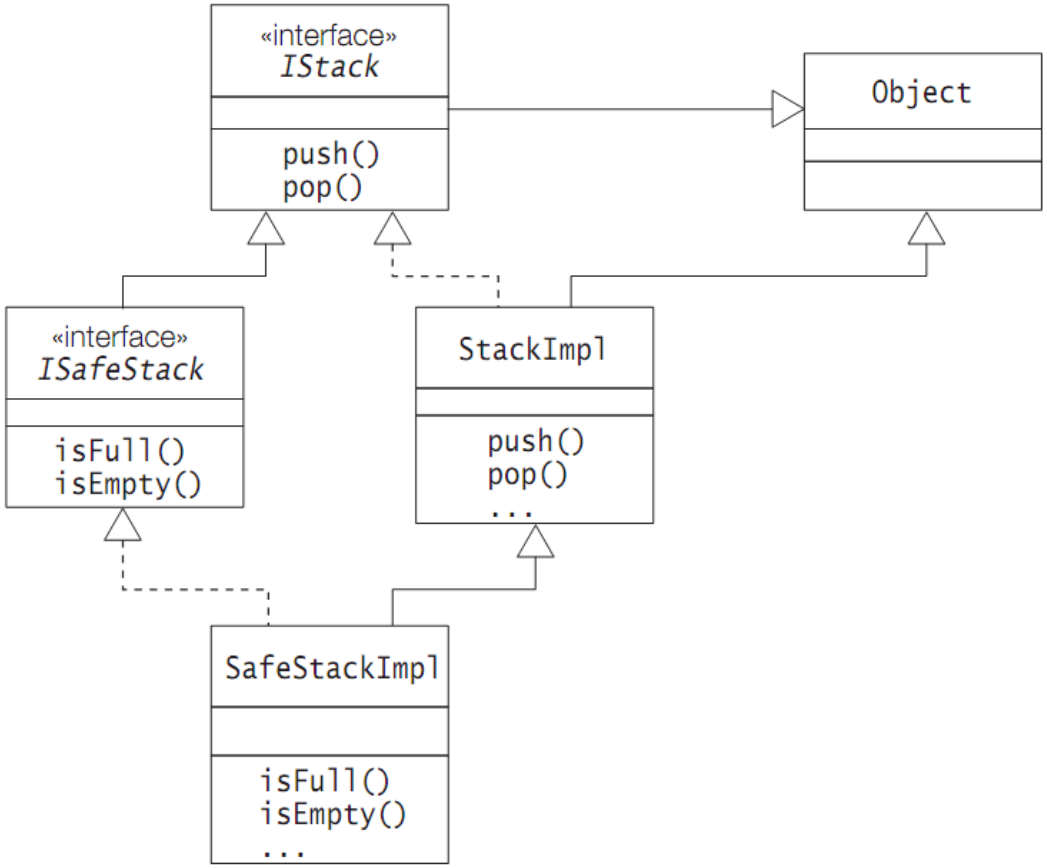
Implementing Interfaces

- A class can neither narrow the accessibility of an interface method nor specify new exceptions in the method's throws clause, as attempting to do so would amount to altering the interface's contract, which is illegal.
- The criteria for overriding methods also apply when implementing interface methods.
- A class can provide implementations of methods declared in an interface, but to reap the benefits of interfaces, the class must also specify the interface name in its implements clause.

Extending Interfaces

- An interface can extend other interfaces, using the extends clause. Unlike extending classes, an interface can extend several interfaces.
- The interfaces extended by an interface (directly or indirectly) are called superinterfaces.
- Conversely, the interface is a subinterface of its superinterfaces.
- Since interfaces define new reference types, superinterfaces and subinterfaces are also supertypes and subtypes, respectively.

Interfaces


«interface»
IStack
push()
pop()
Object
«interface»
ISafeStack
isFull()
isEmpty()
StackImpl
push()
pop()
...
SafeStackImpl
isFull()
isEmpty()
...

Classes and Interfaces Inheritance relations

1. Single implementation inheritance hierarchy between classes: a class extends another class (subclasses–superclasses).
2. Multiple inheritance hierarchy between interfaces: an interface extends other interfaces (subinterfaces–superinterfaces).
3. Multiple interface inheritance hierarchy between interfaces and classes: a class implements interfaces.

Interface References

- Although interfaces cannot be instantiated, references of an interface type can be declared.
- The reference value of an object can be assigned to references of the object's supertypes.
- The reference value of the object is assigned to references of all the object's supertypes, which are used to manipulate the object.

Constants in Interfaces

- An interface can also define named constants.
- Such constants are defined by field declarations and are considered to be public, static, and final.
- These modifiers can be omitted from the declaration.
- Such a constant must be initialized with an initializer expression

Constants in Interfaces

- 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.
- However, if a client is a class that implements this interface or an interface that extends this interface, then the client can also access such constants directly by their simple names, without resorting to the fully qualified name.
- Such a client inherits the interface constants.

Constants in Interfaces

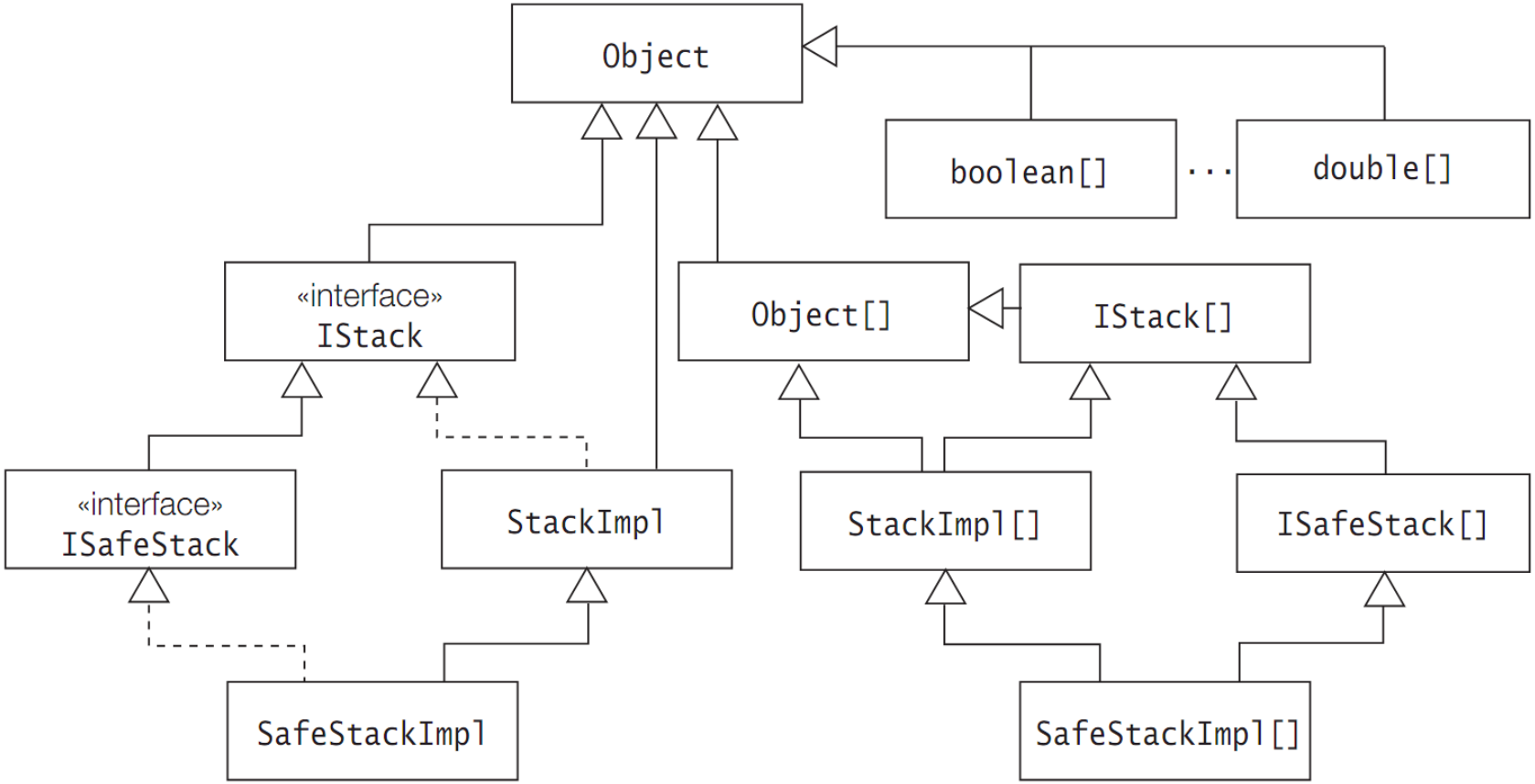
- Extending an interface that has constants is analogous to extending a class having static variables. In particular, these constants can be hidden by the subinterfaces.
- In the case of multiple inheritance of interface constants, any name conflicts can be resolved by using fully qualified names for the constants involved.
- When defining a set of related constants, the recommended practice is to use an enumerated type, rather than named constants in an interface.

Arrays and Subtyping

Only primitive data and reference values can be stored in variables. Only class and array types can be explicitly instantiated to create objects.

Types	Values
Primitive data types	Primitive data values
Class, interface, enum, and array types (*reference types*)	Reference values

Arrays and Subtype Covariance



Arrays and Subtype Covariance

- All reference types are subtypes of the Object type. This applies to classes, interfaces, enum, and array types, as these are all reference types.
- All arrays of reference types are also subtypes of the array type Object[], but arrays of primitive data types are not. Note that the array type Object[] is also a subtype of the Object type.
- If a non-generic reference type is a subtype of another non-generic reference type, the corresponding array types also have an analogous subtype-supertype relationship. This is called the subtype covariance relationship. This relationship however does not hold for parameterized types.
- There is no subtype-supertype relationship between a type and its corresponding array type.

Array Store Check

- An array reference exhibits polymorphic behavior like any other reference, subject to its location in the type hierarchy.
- However, a runtime check is necessary when objects are inserted in an array, as the following example illustrates.

Array Store Check

The following assignment is valid, as a supertype reference (StackImpl[]) can refer to objects of its subtype (SafeStackImpl[]):

```
StackImpl[] stackImplArray = new SafeStackImpl[2]; // (1)
```

Since StackImpl is a supertype of SafeStackImpl, the following assignment is also valid:

```
stackImplArray[0] = new SafeStackImpl(10);          // (2)
```

The assignment at (2) inserts a SafeStackImpl object in the SafeStackImpl[] object (i.e., the array of SafeStackImpl) created at (1).

Array Store Check

Since the type of stackImplArray[i], (0 d i< 2), is StackImpl, it should be possible to do the following assignment as well:

```
stackImplArray[1] = new StackImpl(20);  // (3)
                                        //ArrayStoreException
```

At compile time there are no problems, as the compiler cannot deduce that the array variable stackImplArray will actually denote a SafeStackImpl[] object at runtime.

However, the assignment at (3) results in an ArrayStoreException to be thrown at runtime, as a SafeStackImpl[] object cannot possibly contain objects of type StackImpl.

Reference Values and Conversions

A review of Operators and Expressions on conversions is recommended before proceeding with this section.

Reference values, like primitive values, can be assigned, cast, and passed as arguments.

Conversions can occur in the following contexts:

- assignment
- method invocation
- casting

The Rule of Thumb

- The rule of thumb for the primitive data types is that widening conversions are permitted, but narrowing conversions require an explicit cast.
- The rule of thumb for reference values is that widening conversions up the type hierarchy are permitted, but narrowing conversions down the hierarchy require an explicit cast.

Reference Value Assignment Conversions

In the context of assignments, the following conversions are permitted :

- widening primitive and reference conversions (long <- int, Object <- String)
- boxing conversion of primitive values, followed by optional widening reference conversion (Integer <- int, Number <- Integer <- int)
- unboxing conversion of a primitive value wrapper object, followed by optional widening primitive conversion (long <- int <- Integer).

And only for assignment conversions, we have the following:

- narrowing conversion for constant expressions of non-long integer type, with optional boxing (Byte <- byte <- int)

Reference Value Assignment Conversions

```
Object obj = "Up the tree";
// Widening reference conversion: Object <-- String
String str1 = obj;
// Not ok. Narrowing reference conversion requires a cast
String str2 = new Integer(10);
// Illegal. No relation between String and Integer.
Integer iRef = 10;  // Only boxing
Number num = 10L;
// Boxing, followed by widening: Number <-- Long <-- long
Object obj = 100;
// Boxing, followed by widening: Object <-- Integer <-- int
```

Reference Value Assignment Conversions

The rules for reference value assignment are stated, based on the following code:

```
SourceType srcRef;
// srcRef is appropriately initialized.
DestinationType destRef = srcRef;
```

Reference Value Assignment Conversions

- If the SourceType is a class type, the reference value in srcRef may be assigned to the destRef reference, provided the DestinationType is one of the following:
  - DestinationType is a superclass of the subclass SourceType.
  - DestinationType is an interface type that is implemented by the class SourceType.

Reference Value Assignment Conversions

- If the SourceType is an interface type, the reference value in srcRef may be assigned to the destRef reference, provided the DestinationType is one of the following:
  - DestinationType is Object.
  - DestinationType is a superinterface of subinterface SourceType

Reference Value Assignment Conversions

- If the SourceType is an array type, the reference value in srcRef may be assigned to the destRef reference, provided the DestinationType is one of the following:
  - DestinationType is Object.
  - DestinationType is an array type, where the element type of the SourceType is assignable to the element type of the DestinationType

Overloaded Method Resolution

- How the compiler determines which overloaded method will be invoked by a given method call at runtime?
- Resolution of overloaded methods selects the most specific method for execution.

Overloaded Method Resolution

- One method is more specific than another method if all actual parameters that can be accepted by the one can be accepted by the other. If there is more than one such method, the call is ambiguous.
- The following overloaded methods illustrate this situation:

Overloaded Method Resolution

```
private static void flipFlop(String str, int i, Integer iRef) { // (1)
    out.println(str + " ==> (String, int, Integer)");
}
private static void flipFlop(String str, int i, int j) {        // (2)
    out.println(str + " ==> (String, int, int)");
}
```

Their method signatures are, as follows:

```
flipFlop(String, int, Integer)          // See (1) above
flipFlop(String, int, int)              // See (2) above
```

The following method call is ambiguous:

```
flipFlop("(String, Integer, int)", new Integer(4), 2004);
// (3) Ambiguous call.
```

It has the call signature:

```
flipFlop(String, Integer, int)           // See (3) above
```

Choosing the Most Specific Method

```
class Light { /* ... */ }
class TubeLight extends Light { /* ... */ }
public class Overload {
  boolean testIfOn(Light aLight)          { return true; }    // (1)
  boolean testIfOn(TubeLight aTubeLight) { return false; }   // (2)
  public static void main(String[] args) {
    TubeLight tubeLight = new TubeLight();
    Light     light     = new Light();
    Overload client = new Overload();
    System.out.println(client.testIfOn(tubeLight));// (3) ==> method at (2)
    System.out.println(client.testIfOn(light));    // (4) ==> method at (1)
  }
}
```

Choosing the Most Specific Method

The algorithm used by the compiler for the resolution of overloaded methods incorporates the following phases:

1. It first performs overload resolution without permitting boxing, unboxing, or the use of a varargs call.
2. If phase (1) fails, it performs overload resolution allowing boxing and unboxing, but excluding the use of a varargs call.
3. If phase (2) fails, it performs overload resolution combining a varargs call, boxing, and unboxing.

```
import static java.lang.System.out;
class OverloadResolution {
  public void action(String str) {          // (1)
    String signature = "(String)";
    out.println(str + " => " + signature);
  }
  public void action(String str, int m) {           // (2)
    String signature = "(String, int)";
    out.println(str + " => " + signature);
  }
  public void action(String str, int m, int n) {    // (3)
    String signature = "(String, int, int)";
    out.println(str + " => " + signature);
  }
  public void action(String str, Integer... data) { // (4)
    String signature = "(String, Integer[])";
    out.println(str + " => " + signature);
  }
  public void action(String str, Number... data) {  // (5)
    String signature = "(String, Number[])";
    out.println(str + " => " + signature);
  }
  public void action(String str, Object... data) {  // (6)
    String signature = "(String, Object[])";
    out.println(str + " => " + signature);
  }
```

```
 public static void main(String[] args) {
    OverloadResolution ref = new OverloadResolution();
    ref.action("(String)");                              // (8)  calls (1)
    ref.action("(String, int)",           10);           // (9)  calls (2)
    ref.action("(String, Integer)", new Integer(10));    // (10) calls (2)
    ref.action("(String, int, byte)", 10, (byte)20);     // (11) calls (3)
    ref.action("(String, int, int)",      10,  20);      // (12) calls (3)
    ref.action("(String, int, long)",     10,  20L);     // (13) calls (5)
    ref.action("(String, int, int, int)", 10,  20,  30);  // (14) calls (4)
    ref.action("(String, int, double)",   10,  20.0);      // (15) calls (5)
    ref.action("(String, int, String)",   10,  "what?");   // (16) calls (6)
    ref.action("(String, boolean)",       false);          // (17) calls (6)
  }
}
```

Reference Casting

The type cast expression for reference types has the following syntax:

(<destination type>) <reference expression>

The following conversions can be applied to the operand of a cast operator:

- both widening and narrowing reference conversions, followed optionally by an unchecked conversion
- both boxing and unboxing conversions

Reference Casting

Boxing and unboxing conversions that can occur during casting is illustrated by the following code:

```
// (1) Boxing and casting: Number <- Integer <- int:
Number num = (Number) 100;
// (2) Casting, boxing, casting: Object <- Integer <- int <- double:
Object obj = (Object) (int) 10.5;
// (3) Casting, unboxing, casting: double <- int <- Integer <- Object:
double d = (double) (Integer) obj;
```

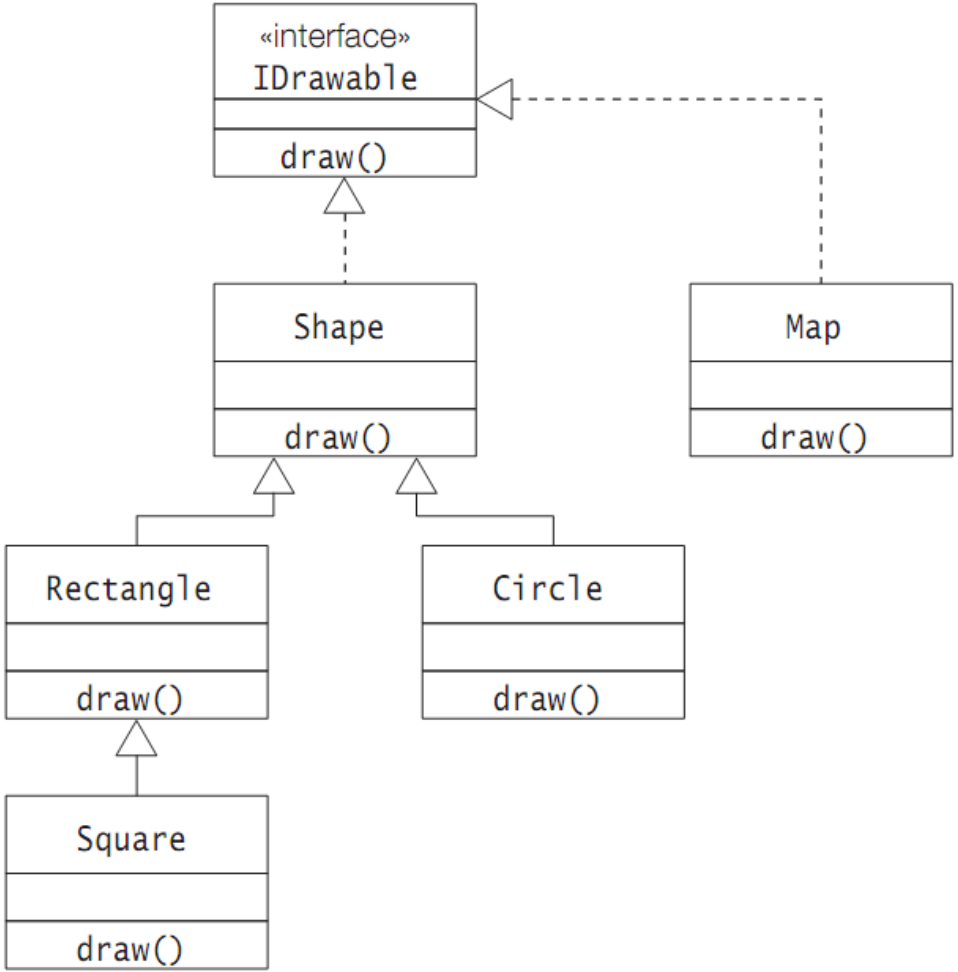
The `instanceof` Operator

The binary instanceof operator can be used for comparing types. It has the following syntax (*note that the keyword is composed of only lowercase letters*):

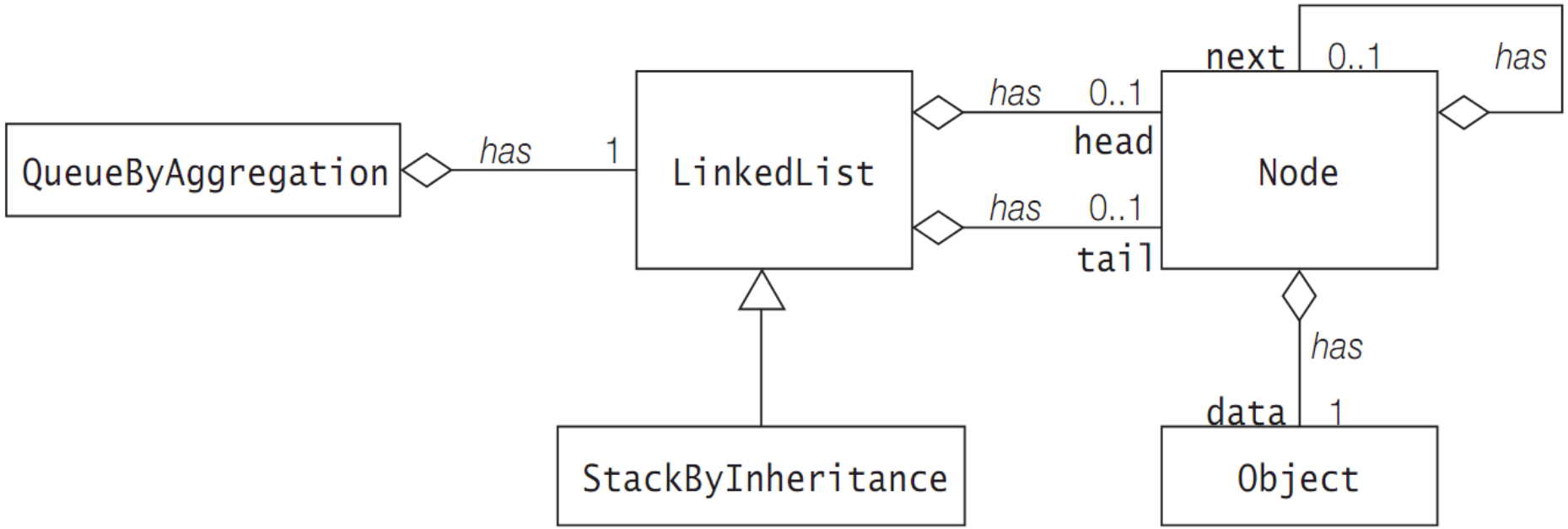
<reference expression> **instanceof** <destination type>

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 objects of different types at different times during execution.
- A supertype reference exhibits polymorphic behavior since it can denote objects of its subtypes.


«interface»
IDrawable
draw()
Shape
draw()
Map
draw()
Rectangle
draw()
Circle
draw()
Square
draw()

Inheritance Versus Aggregation



Inheritance Versus 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 thelifetime of the objects involved; otherwise, aggregation is the best choice.

Inheritance Versus Aggregation

A ***role*** is often confused with an **is-a** relationship. For example, given the class Employee, it would not be a good idea to model the roles an employee can play (such as a manager or a cashier) by inheritance if these roles change intermittently.

Changing roles would involve a new object to represent the new role every time this happens.

Basic Concepts in Object-Oriented Design

- Encapsulation
- Cohesion
- Coupling

Encapsulation

Encapsulation is achieved through information hiding, by making judicious use of language features provided for this purpose. Information hiding in Java can be achieved at different levels of granularity:

- method or block level
- class level
- package level

Cohesion

Cohesion is an inter-class measure of how well-structured and closely-related the functionality is in a class.

The objective is to design classes with high cohesion, that perform well-defined and related tasks (also called functional cohesion).

The public methods of a highly cohesive class typically implement a single specific task that is related to the purpose of the class.

Coupling

- Coupling is a measure of intra-class dependencies. Objects need to interact with each other, therefore dependencies between classes are inherent in OO design.
- However, these dependencies should be minimized in order to achieve loose coupling, which aids in creating extensible applications.

High cohesion and loose coupling help to achieve the main goals of OO design: maintainability, reusability, extensibility, and reliability

Object-Oriented Programming

That’s all folks!