Chapter 7
Object-Oriented Programming (OOP)

Reference:

A Programmer's Guide to Java SE 8 Oracle Certified Associate (OCA),
Khalid A. Mughal, Rolf W. Rasmussen:
Addison-Wesley Professional, 2016, ISBN: 0132930218
(http://www.ii.uib.no/~khalid/ocajp8/)

Khalid Azim Mughal
Associate Professor Emeritus
Department of Informatics
University of Bergen, Norway.
khalid.mughal@uib.no
http://www.ii.uib.no/~khalid

Version date: 2018-11-06

Overview

- Single Implementation Inheritance
- Overriding Methods
- Hiding Members
- The Object Reference super
- Chaining Constructors Using this() and super()
- Arrays and Subtyping

Single Implementation Inheritance

- *Inheritance* promotes *code reuse* in OOP.
- It allows new classes to be *derived* from existing ones.
- 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.
- *Implementation inheritance* (a.k.a. *class inheritance*) is achieved by extending classes (i.e., adding new fields and methods) and modifying inherited members.

Inheritance and 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 (Example 7.1).

```
class TubeLight extends Light { ... }   // TubeLight is a subclass of Light.
```

- Using appropriate accessibility modifiers, the superclass can limit which members can be accessed directly and, thereby, inherited by its subclasses.
- Since constructors are *not* members of a class, they are *not* inherited by a subclass.

Example 7.1 Extending Classes: Inheritance and Accessibility

```
// File: Utility.java
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

  // No-argument 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) Subclass uses the extends clause.
  // Instance fields:
  private int tubeLength = 54;
  private int colorNo    = 10;

  // Instance methods:
  public int getTubeLength() { return tubeLength; }

  public void printInfo() {
    System.out.println("From the subclass:");
    System.out.println("Tube length: "  + tubeLength);
```

```
    System.out.println("Color number: " + colorNo);
    System.out.println("Tube length: "  + getTubeLength());
    System.out.println();
    System.out.println("From the superclass:");
    System.out.println("Wattage: "      + noOfWatts);     // Inherited.
//  System.out.println("Indicator: "    + indicator);     // Not inherited.
    System.out.println("Location: "     + location);      // Inherited.
//  System.out.println("Counter: "   + counter);          // Not inherited.
    switchOn();                                          // Inherited
    switchOff();                                         // Inherited
    System.out.println("Indicator: "    + isOn());        // Inherited.
//  printLocation();                                     // Not inherited.
    writeCount();                                        // Inherited.
  }
  // ...
}
//_____________________________________________________________________________
public class Utility {                    // (3)
  public static void main(String[] args) {
    new TubeLight().printInfo();
  }
}
```

- Output from the program:

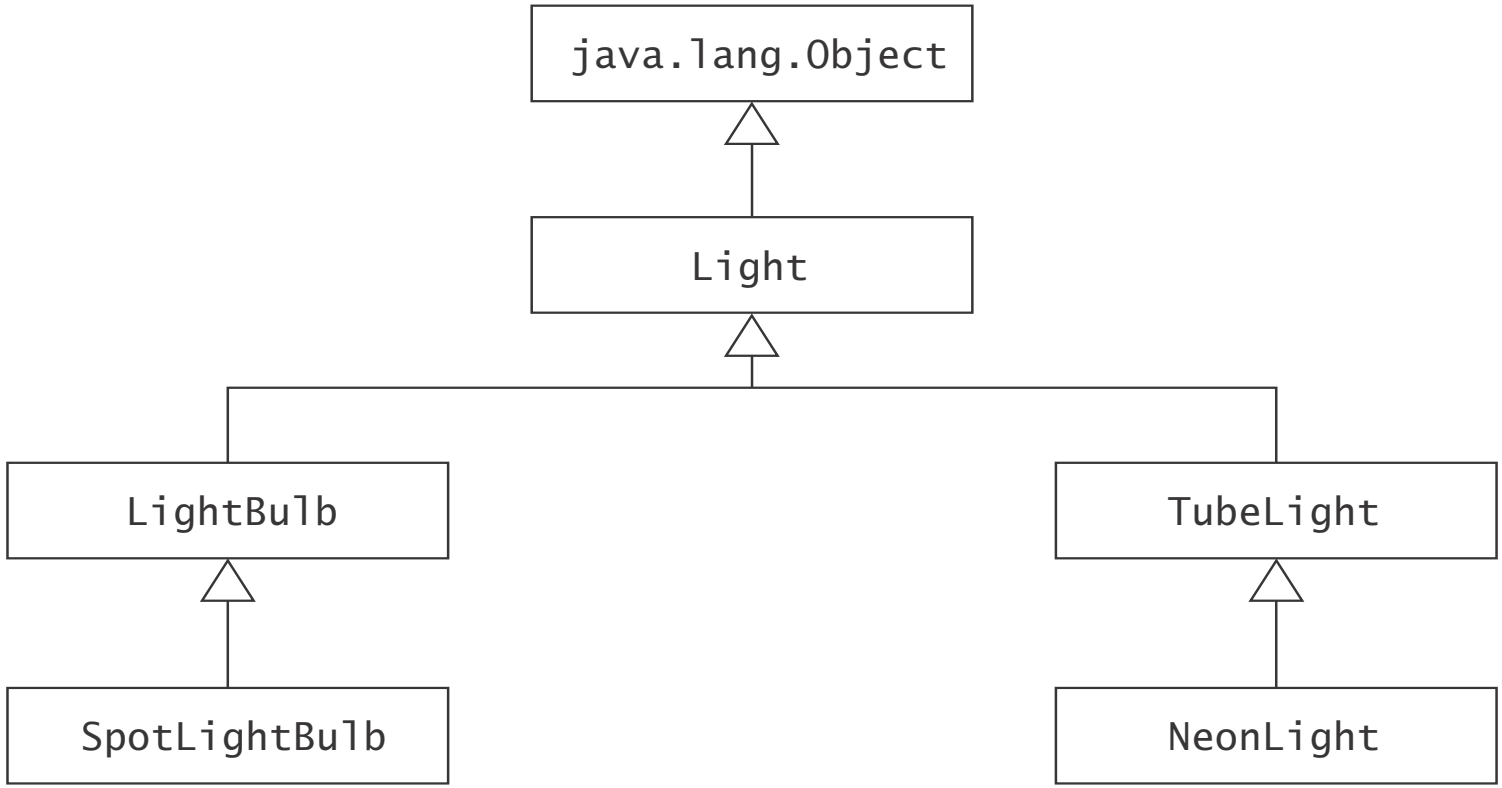
```
From the subclass:
Tube length: 54
Color number: 10
Tube length: 54

From the superclass:
Wattage: 50
Location: X
Indicator: false
Number of lights: 1
```

Inheritance Hierarchy

- In Java, a class can extend only *one* class; i.e., it can only have one immediate superclass.
  - This kind of inheritance is sometimes called *single* or *linear implementation inheritance*.
  - The inheritance relationship can be depicted as an *inheritance hierarchy* (also called *class hierarchy*).
  - Classes higher up in the hierarchy are more *generalized* (often called *broader*), as they abstract the class behavior.
  - Classes lower down in the hierarchy are more *specialized* (often called *narrower*), as they customize the inherited behavior by additional properties and behavior.
  - The `java.lang.Object` class is always at the top (a.k.a. *root*) of any Java inheritance hierarchy, as all classes, with the exception of the `Object` class itself, inherit (either directly or indirectly) from this class.

Figure 7.1 Inheritance Hierarchy



Relationships: is-a

- Inheritance defines the relationship *is-a* (also called the *superclass–subclass* relationship) between a superclass and its subclasses.
- This means that an object of a subclass *is-a* superclass object, and can be used wherever an object of the superclass can be used.
- Litmus test for choosing inheritance in object-oriented design.
- The inheritance relationship is transitive: if class `B` extends class `A` and class `C` extends class `B`, then class `C` will also inherit from class `A` via class `B`.
- The *is-a* relationship does not hold between peer classes.

Relationships: has-a

- *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.
- A composite object can only store *reference values* of its constituent objects in its fields.
- This relationship defines an *aggregation hierarchy* (also called *object hierarchy*) that embodies the *has-a* relationship.
- Constituent objects can be shared between objects, and if their lifetimes are dependent on the lifetime of the composite object, then this relationship is called *composition*, and implies strong ownership of the parts by the composite object.

The Supertype-Subtype Relationship

- A class defines a *reference type*, i.e., a data type whose objects can only be accessed by references.
- Therefore the inheritance hierarchy can be regarded as a *type hierarchy*, embodying the *supertype-subtype relationship* between reference types.
- 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.

```
Light light = new TubeLight();          // (1) widening reference conversion
light.switchOn();                       // (2)
light.getTubeLength();                  // (3) Not OK.
```

- The compiler only knows about the *declared type* (a.k.a. *static 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 refers to at runtime that determines which method is executed.

- The type of the object that the reference refers to at runtime is often called the *dynamic type* of the reference.
- Eliciting subtype-specific behavior using a supertype reference requires a narrowing reference conversion with an explicit cast.

Overriding Methods

Instance Method Overriding

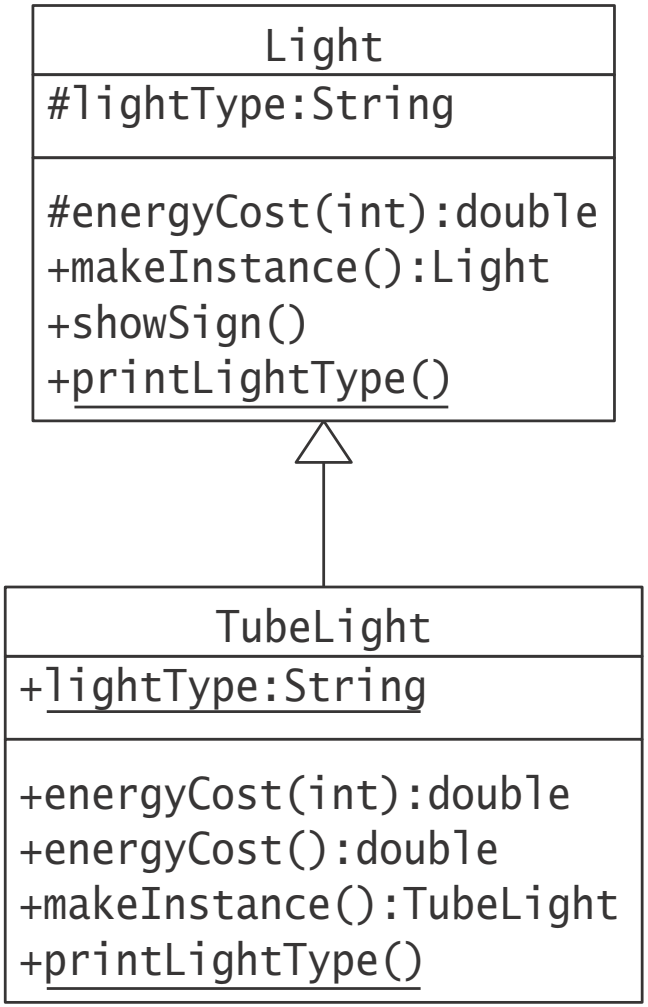
- Under certain circumstances, a subclass can *override instance methods* that it inherits from its superclass.
- Overriding such a method allows the subclass to provide its *own* implementation of the method.
- The overridden method in the superclass is *not* inherited by the subclass.
- When the method is invoked on an object of the subclass, it is the method implementation in the subclass that is executed.

Overriding Rules

1. The new method definition in the subclass 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 of the superclass.
   – A method's signature does not comprise the `final` modifier of parameters.
2. The return type of the overriding method can be a *subtype* of the return type of the overridden method (called *covariant return*).
3. The new method definition cannot *narrow* the accessibility of the method, but it can *widen* it.
4. 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.

- These requirements also apply to *interfaces*.
- The `@Override` annotation should be used on the overriding method in the subclass.
  – The compiler will report an error if the method definition does *not* override an inherited method.

Figure 7.2 Inheritance Hierarchy



Accessibility:
`+ : public`
`- : private`
`# : protected`
None `: package`

Example 7.2 Overriding, Overloading, and Hiding

```
// File: Client2.java
//Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String lightType = "Generic Light";   // (1) Instance field

  protected double energyCost(int noOfHours)      // (2) Instance method
      throws InvalidHoursException {
    System.out.print(">> Light.energyCost(int): ");
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double cost = 00.20 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public Light makeInstance() {                   // (3) Instance method
```

```
    System.out.print(">> Light.makeInstance(): ");
    return new Light();
  }

  public void showSign() {                          // (4) Instance method
    System.out.print(">> Light.showSign(): ");
    System.out.println("Let there be light!");
  }

  public static void printLightType() {             // (5) Static method
    System.out.print(">> Static Light.printLightType(): ");
    System.out.println("Generic Light");
  }
}
//_____________________________________________________________________________
class TubeLight extends Light {

  public static String lightType = "Tube Light";  // (6) Hiding field at (1).

  @Override
  public double energyCost(final int noOfHours)   // (7) Overriding instance
      throws ZeroHoursException {                 //     method at (2).
    System.out.print(">> TubeLight.energyCost(int): ");
```

```
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double cost = 00.10 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public double energyCost() {          // (8) Overloading method at (7).
    System.out.print(">> TubeLight.energyCost(): ");
    double flatrate = 20.00;
    System.out.println("Energy cost for " + lightType + ": " + flatrate);
    return flatrate;
  }

  @Override
  public TubeLight makeInstance() {     // (9) Overriding instance method at (3).
    System.out.print(">> TubeLight.makeInstance(): ");
    return new TubeLight();
  }

  public static void printLightType() { // (10) Hiding static method at (5).
    System.out.print(">> Static TubeLight.printLightType(): ");
    System.out.println(lightType);
```

```
  }
}
//_____________________________________________________________________________
public class Client2 {
  public static void main(String[] args)      // (11)
      throws InvalidHoursException {

    TubeLight tubeLight = new TubeLight();    // (12)
    Light     light1    = tubeLight;          // (13) Aliases.
    Light     light2    = new Light();        // (14)

    System.out.println("Invoke overridden instance method:");
    tubeLight.energyCost(50);                 // (15) Invokes method at (7).
    light1.energyCost(50);                    // (16) Invokes method at (7).
    light2.energyCost(50);                    // (17) Invokes method at (2).

    System.out.println(
        "\nInvoke overridden instance method with covariant return:");
    System.out.println(
        light2.makeInstance().getClass());    // (18) Invokes method at (3).
    System.out.println(
        tubeLight.makeInstance().getClass()); // (19) Invokes method at (9).
```

```
        System.out.println("\nAccess hidden field:");
        System.out.println(tubeLight.lightType);  // (20) Accesses field at (6).
        System.out.println(light1.lightType);     // (21) Accesses field at (1).
        System.out.println(light2.lightType);     // (22) Accesses field at (1).

        System.out.println("\nInvoke hidden static method:");
        tubeLight.printLightType();               // (23) Invokes method at (10).
        light1.printLightType();                  // (24) Invokes method at (5).
        light2.printLightType();                  // (25) Invokes method at (5).

        System.out.println("\nInvoke overloaded method:");
        tubeLight.energyCost();                   // (26) Invokes method at (8).
    }
}
```

- Output from the program:

```
Invoke overridden instance method:
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> Light.energyCost(int): Energy cost for Generic Light: 10.0

Invoke overridden instance method with covariant return:
>> Light.makeInstance(): class Light
```

```
>> TubeLight.makeInstance(): class TubeLight

Access hidden field:
Tube Light
Generic Light
Generic Light

Invoke hidden static method:
>> Static TubeLight.printLightType(): Tube Light
>> Static Light.printLightType(): Generic Light
>> Static Light.printLightType(): Generic Light

Invoke overloaded method:
>> TubeLight.energyCost(): Energy cost for Tube Light: 20.0
```

Some More Facts about Overriding

- A subclass must use the keyword `super` in order to invoke an overridden method in the superclass.
- A `final` method cannot be overridden, because the modifier `final` prevents method overriding.
- A `private` method cannot be overridden.
- An `abstract` method forces the non-abstract subclasses to override the method, in order to provide an implementation.
- A subclass within the *same* package as the superclass can override any non-`final` and non-`private` methods declared in the superclass.
- A subclass in a *different* package can override only the non-`final` methods that are declared either `public` or `protected` in the superclass.
- An instance method in a subclass cannot override a `static` method in the superclass.
  - A static method in a subclass can *hide* a `static` method in the superclass.
- Constructors, since they are not methods, cannot be overridden.

Covariant `return` in Overriding Methods

- The method signatures are the same, but the return type in the overriding method is a subtype of the return type of the overridden method.
- Covariant return only applies to *reference* types, not to primitive types.

Method Overloading

- Overloading occurs when the method names are the same, but the parameter lists differ.
  - The parameters must differ either in type, order, or number.
  - As the return type is not a part of the method signature, just having different return types is not enough to overload methods.
- Both instance and static methods can be overloaded in the class they are defined in or in a subclass of their class.
- If the right kinds of arguments are passed in the method call occurring in the subclass, the overloaded method in the superclass will be invoked.
- For overloaded methods, which method implementation will be executed at run-time is determined at *compile time*, but for overridden methods, the method implementation to be executed is determined at *runtime*.

Table 7.1 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 be restrictive about exceptions thrown.	Can be different.
Accessibility	Can make it less restrictive, but not more restrictive.	Can be different.
`final` modifier	Cannot be overridden.	Can be overloaded in the same class or in a subclass.
Declaration context	An instance method can only be overridden in a subclass.	An instance or static method can be overloaded in the same class or in a subclass.
Method call resolution	The *dynamic 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.

Field Hiding

- A subclass cannot override inherited fields of the superclass, but it can *hide* them.
- The subclass can define fields with the same name as in the superclass.
- Hidden fields in the superclass cannot be accessed in the subclass by their simple names; therefore, they are not inherited by the subclass.
- A hidden static field can always be invoked by using the superclass name in the subclass declaration.
- Additionally, the keyword `super` can be used in non-static code in the subclass declaration to access hidden static fields.
- When a field of an object is accessed using a reference, it is the *declared type* of the reference, not the type of the current object denoted by the reference, that determines which field will actually be accessed.
- In contrast to method overriding, where an instance method cannot override a static method, there are no such restrictions on the hiding of fields.
- The declared type of the fields need not be the same either, it is only the field name that matters in the hiding of fields.

Static Method Hiding

- A static method in a subclass *cannot* override an instance method from the superclass, but it can *hide* a *static* method from the superclass if the exact requirements for overriding instance methods are fulfilled.
- A hidden superclass static method is not inherited.
  - 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`).
- Analogous to accessing fields, the static method invoked at runtime is determined by the *declared type* of the reference.
- Analogous to hidden fields, a hidden static method can always be invoked by using the superclass name or by using the keyword `super` in non-static code in the subclass declaration.

Table 7.2 Same Signature for Subclass and Superclass Method

Subclass method has the same signature as the superclass method	Instance method in superclass	Static method in superclass
Instance method in subclass	Overriding	Compile-time error
Static method in subclass	Compile-time error	Hiding

The Object Reference `super`

- The `this` reference can be used in non-static code to refer to the current object.
- The keyword `super`, on the other hand, can be used in non-static code 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.
- In method invocations with `super`, the method from the superclass is invoked regardless of the actual type of the current object or whether the current class overrides the method.
- It is 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.

Example 7.3 Using the `super` *Keyword*

```
// File: Client3.java
//Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}

class Light {

  protected String lightType = "Generic Light";   // (1) Instance field

  protected double energyCost(int noOfHours)      // (2) Instance method
      throws InvalidHoursException {
    System.out.print(">> Light.energyCost(int): ");
    if (noOfHours < 0)
      throw new NegativeHoursException();
    double cost = 00.20 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public Light makeInstance() {                   // (3) Instance method
```

```
    System.out.print(">> Light.makeInstance(): ");
    return new Light();
  }

  public void showSign() {                          // (4) Instance method
    System.out.print(">> Light.showSign(): ");
    System.out.println("Let there be light!");
  }

  public static void printLightType() {             // (5) Static method
    System.out.print(">> Static Light.printLightType(): ");
    System.out.println("Generic Light");
  }
}
//_____________________________________________________________________________
class TubeLight extends Light {

  public static String lightType = "Tube Light";  // (6) Hiding field at (1).

  @Override
  public double energyCost(final int noOfHours)   // (7) Overriding instance
      throws ZeroHoursException {                 //     method at (2).
    System.out.print(">> TubeLight.energyCost(int): ");
```

```
    if (noOfHours == 0)
      throw new ZeroHoursException();
    double cost = 00.10 * noOfHours;
    System.out.println("Energy cost for " + lightType + ": " + cost);
    return cost;
  }

  public double energyCost() {          // (8) Overloading method at (7).
    System.out.print(">> TubeLight.energyCost(): ");
    double flatrate = 20.00;
    System.out.println("Energy cost for " + lightType + ": " + flatrate);
    return flatrate;
  }

  @Override
  public TubeLight makeInstance() {     // (9) Overriding instance method at (3).
    System.out.print(">> TubeLight.makeInstance(): ");
    return new TubeLight();
  }

  public static void printLightType() { // (10) Hiding static method at (5).
    System.out.print(">> Static TubeLight.printLightType(): ");
    System.out.println(lightType);
```

```
  }
}
//_____________________________________________________________________________
class NeonLight extends TubeLight {
  // ...
  public void demonstrate()                          // (11)
      throws InvalidHoursException {
    super.showSign();                                // (12) Invokes method at (4)
    super.energyCost(50);                            // (13) Invokes method at (7)
    super.energyCost();                              // (14) Invokes method at (8)

    ((Light) this).energyCost(50);                   // (15) Invokes method at (7)

    System.out.println(super.lightType);             // (16) Accesses field at (6)
    System.out.println(((Light) this).lightType); // (17) Accesses field at (1)

    super.printLightType();                          // (18) Invokes method at (10)
    ((Light) this).printLightType();                 // (19) Invokes method at (5)
  }
}
//_____________________________________________________________________________
public class Client3 {
  public static void main(String[] args)
```

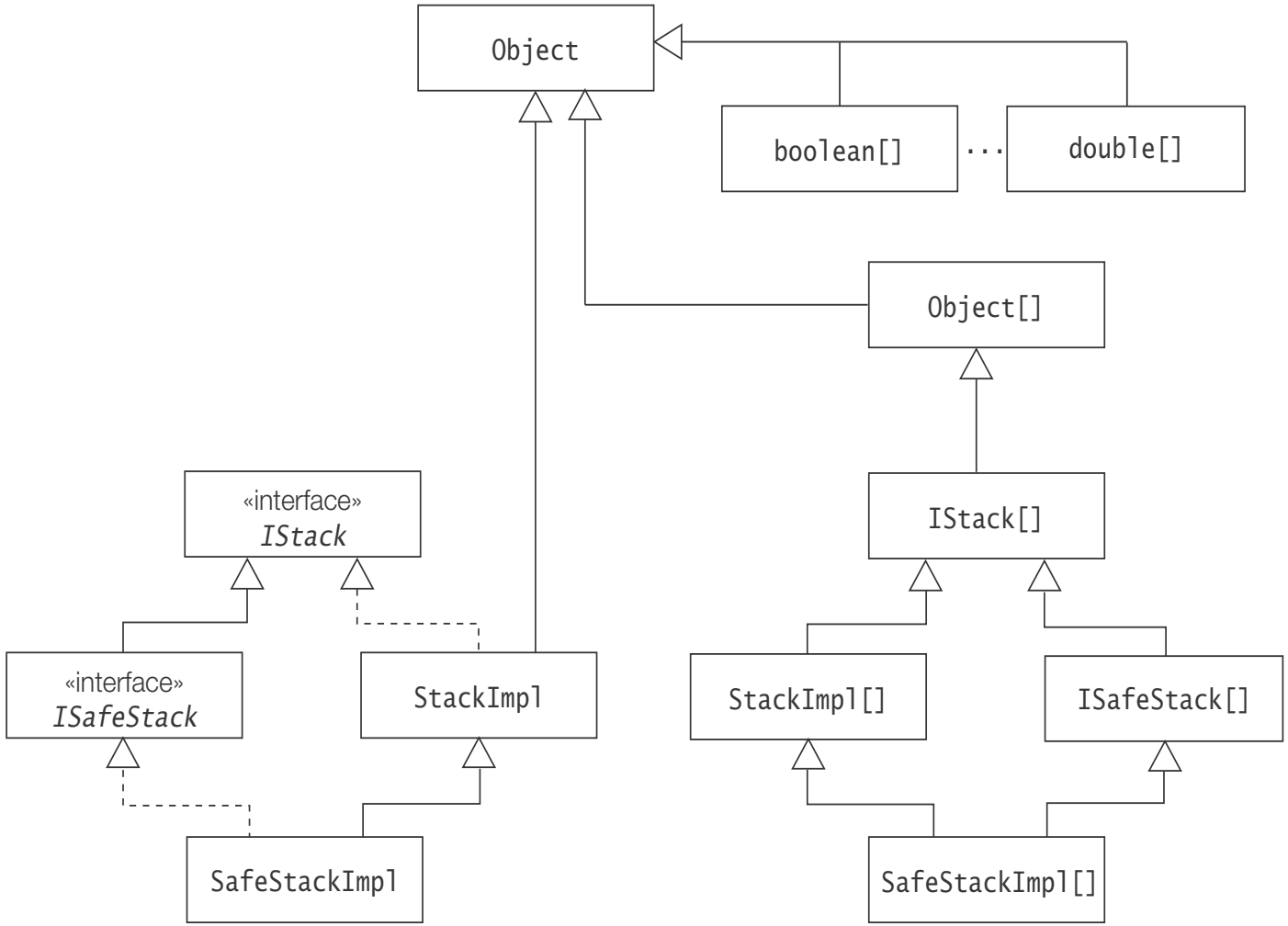
```
        throws InvalidHoursException {
      NeonLight neonRef = new NeonLight();
      neonRef.demonstrate();
    }
}
```

- Output from the program:

```
>> Light.showSign(): Let there be light!
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
>> TubeLight.energyCost(): Energy cost for Tube Light: 20.0
>> TubeLight.energyCost(int): Energy cost for Tube Light: 5.0
Tube Light
Generic Light
>> Static TubeLight.printLightType(): Tube Light
>> Static Light.printLightType(): Generic Light
```

Arrays and Subtype Covariance.

Figure 7.3 Reference Type Hierarchy: Arrays and Subtype Covariance



- From the type hierarchy, we can summarize the following:
  - *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.
  - 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*.
  - There is *no* subtype-supertype relationship between a type and its corresponding array type.
- We can create an array of an interface type, but we cannot instantiate an interface (as is the case with `abstract` classes).

```
ISafeStack[] iSafeStackArray = new ISafeStack[5];   // Initialized to default value null.
```

Array Store Check

- An array reference exhibits polymorphic behavior like any other reference, subject to its location in the type hierarchy.
- A runtime check is necessary when objects are storing a value in an array.
- 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)
```

- Since the type of `stackImplArray[i]`, ($0 \leq$ `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, but an `ArrayStoreException` thrown at runtime.
- In order to make the array store check feasible at runtime, the array retains information about its declared element type at runtime.