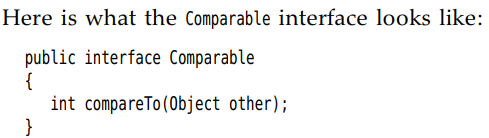
Chapter 6: Interfaces, Lambda Expressions, and Inner Classes

-You need to master these advanced techniques to complete Java tool chest.

# 6.1 Interfaces

## 6.1.1 The Interface concept

-Interface is not a class but a set of requirements for the classes that want to conform to the interface.



+The sort method of Arrays method promises to sort an array of objects but one condition: the objects must belong to classes that implement Comparable interface

+Any class that implements the Comparable interface is required to have a compareTo method.

+Manager also implement Comparable<Employee>. If override, Manager must compare managers to employee. => **ClassCastException**

+Each compareTo method should start with



+If there is a common algorithm for comparing subclass, provide a single compareTo method in superclass and declare it **final**

-All **methods** in **interface** are **automatically** **public**, whether the keyword is specified or not.

-**Interface** is an **abstract class** with **no instance fields**. All fields in Interface are implicitly **public, static and final**, whether the keywords are specified or not.

-**Make** a **class** **implement** an **interface**: declare the class intends to implement, supply definition for all methods in interfaces.

## 6.1.2 Properties of Interfaces

-Interfaces are not classes. You can never use **new**

****

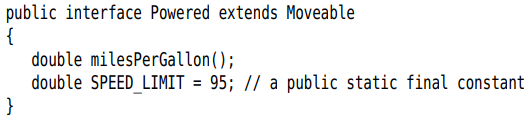
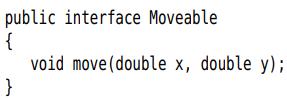
-But you can **declare**: . An interface variable must refer to implemented object: 

-A **reference** of interface type can be **cast** to any class that implements this interface.

-Check whether an object implements an interface



-You can **extend any number of interfaces**.



-An overriding method must return either the **same type** or a **sub type** of the **return type** of overridden method.

-Classes can implement **multiple interfaces**.



+More than one method declaration may be implemented by a singer method declaration if they have the same name.

+If the methods throw exception, the overriding declaration must throw the subclass of the exceptions.

-**Multiple inheritance of state**: inherit instance fields.

-**Multiple inheritance of type**: implement interfaces and extend classes.

## 6.1.3 Interfaces and Abstract classes

-Class only extend one class, but many interfaces.

## 6.1.4 Static and private methods

-Static has a body. You can **redeclare** a static method as a **default** method in sub interface.

-private: You can use private static or instance methods.

## 6.1.5 Default methods

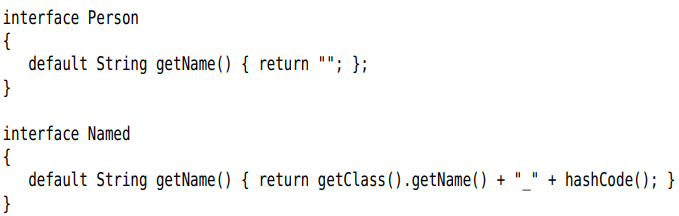
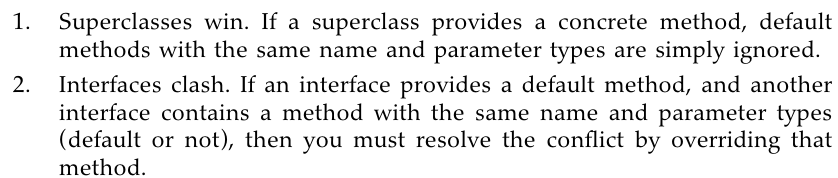
-Default methods supply a **default** implementation with default modifier. It can call other methods.

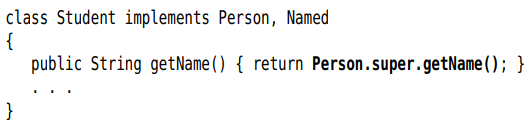
-It has a body.

-A subinterface can redeclare an inherited default method as **abstract** or provide a different implementation.

## 6.1.6 Resolving default method conflicts

-A default method in one interface and have this method in a superclass or interface:





-Extend class and implements interface: only the superclass method matter, the interface is ignored



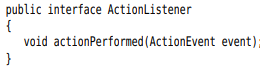
-**Note**: Never make default method that redefines the methods in Object class.

## 6.1.7 Interfaces and callbacks

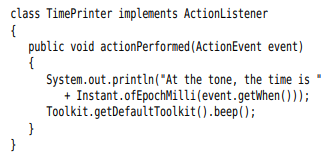
-**Callback**: a common pattern, you should specify the action that should occur whenever a particular event happens. Example: an action to occur when a button clicked, menu item selected.

-Example: java.swing.Timer class: set the time interval and tell what it do whenever the interval has elapsed. You pass an object of some class.

+The timer requires you specify an object of a class that implements java.awt.event.ActionListener interface



+Print a message with a beeb one every second:



+**ActionEvent**: give information about the event, such as the time when the event happened.

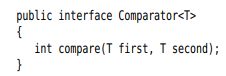
+Construct an object of this object and pass it to Timer:



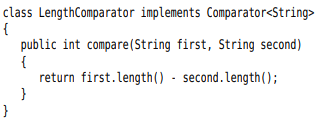


## 6.1.8 The Comparator Interface

-A second version of Arrays.sort method whose parameters are array and *comparator*



+Compare strings by length, define a class that implements Comparator<String>



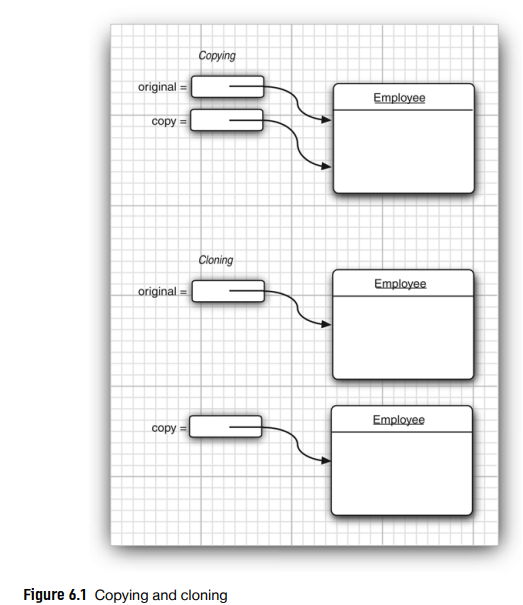
+Do the comparison or sort an array

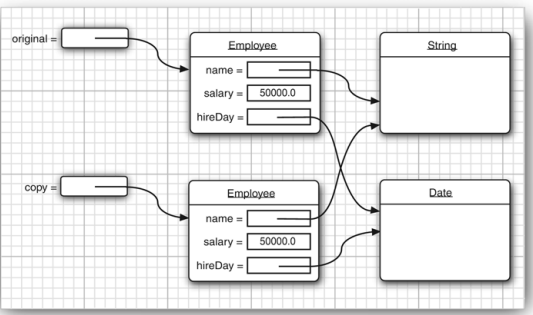




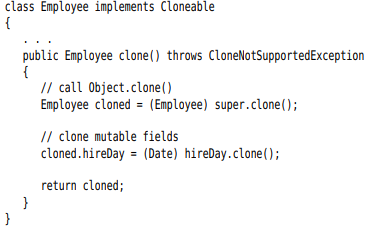
## 6.1.9 Object Cloning

-**Cloneable** interface: **clone** method.





-Deep copy:



+throw CloneNotSupportedException: class does not implement the Cloneable interface.

-Cloning is less common. If clients need to make deep copies, then do it.

-Note: array types has public clone method.



# 6.2 Lambda Expression

## 6.2.1 Why Lambdas

-**Lambda** expression is a block of code that you can pass around so it can be executed later.

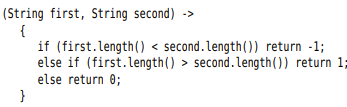
## 6.2.2 The Syntax of Lambda expressions



+A block of code, with the specification of any variable that must be passed to the code.

+**Form**: parameters, ->, expression

-If the code doesn’t fit in a single expression:

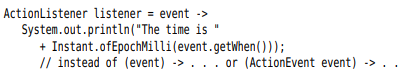


-**No parameter**: 

-**Omit** the **parameter types** that can be **inferred**



-**Single parameter**: can omit the parentheses



-You **never** **specify** the **result type** of lambda expression.

## 6.2.3 Functional Interfaces

-**Function interfaces**: they encapsulate blocks of code (ActionListener, Comparator). You can supply a lambda expression whenever an object of an interface with a single abstract method is expected.

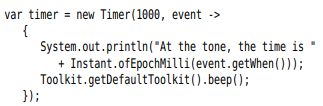
-Arrays.sort:



+2nd parameter requires an instance of Comparator interface with a single method.

+Array.sort receives an object of some class that implement Comparator<String>. Invoking the compare method on that object executes the body of lambda expression.

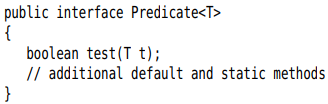
-Think **lambda expression** as a **function** and can be **passed** to a **functional interface**.



+Easier to read than a class that implement ActionListener interface.

-Conversion to functional interface is the *only* thing that you can do with lambda expression in Java.

-A useful interface is java.util.function.**Predicate**





-**Supplier<T>**: yield a value of type T when is called, used for *lazy evaluation*.



## 6.2.4 Method references

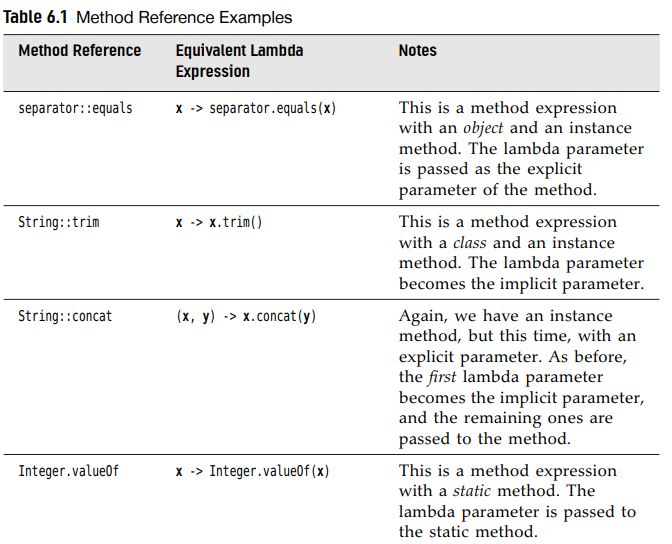
-Lambda expression involves a **single method**:

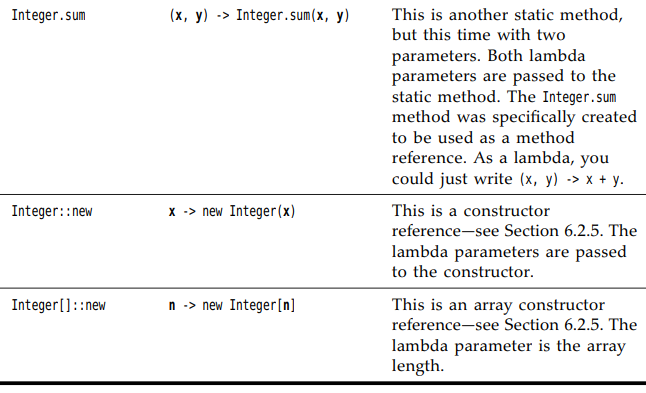


-> **method reference**: 

+It directs the compiler to produce an instance of a functional interface, overriding the single abstract method of the interface

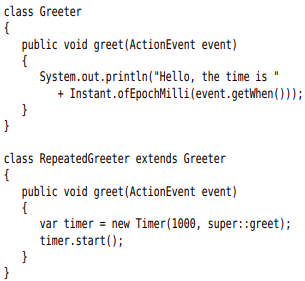
-**Method reference example**:



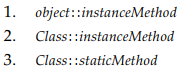


-You can capture **this** parameter and **super** in method reference:

 = 

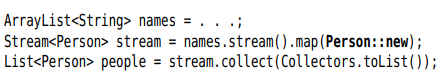






## 6.2.5 Constructor reference

-Constructor reference are like method references, but the name of method is **new.**



+Turn the list of strings into an array of Person objects.

-You can form constructor references with array types:

**int[]::new** (parameter: the length of array). It’s useful to overcome a limitation of java.

* = 

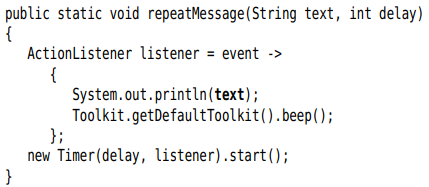
-**Stream.toArray()** return Object array, we can pass constructor reference to this method:





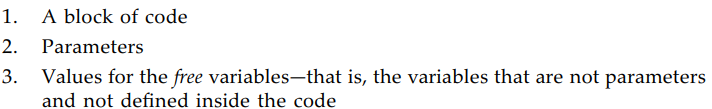
## 6.2.6 Variable scope

-Access variables from enclosing method or class in lambda expression:



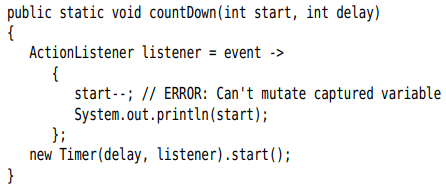
+**text** variable is not defined in lambda expression

-A lambda expression has 3 ingredients:



-The free variable **text** have been **captured** (stored) by lambda expression. The block of code together with values of free variable is a *closure.*

-An important restriction: only reference variable whose value doesn’t change.



-Any captured variable in lambda expression must be **effectively final**. This variable is never assigned a new value after it has been initialized.

-The body of lambda expression has *the same scope as a nested block*.

## 6.2.7 Processing lambda expressions

-The point of using lambdas is *deferred execution*.

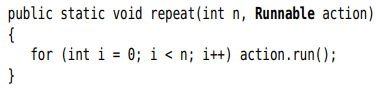
-We write methods that can consume lambda expressions

-Example: repeat an action in n times

+The action and count are passed to a **repeat** method:

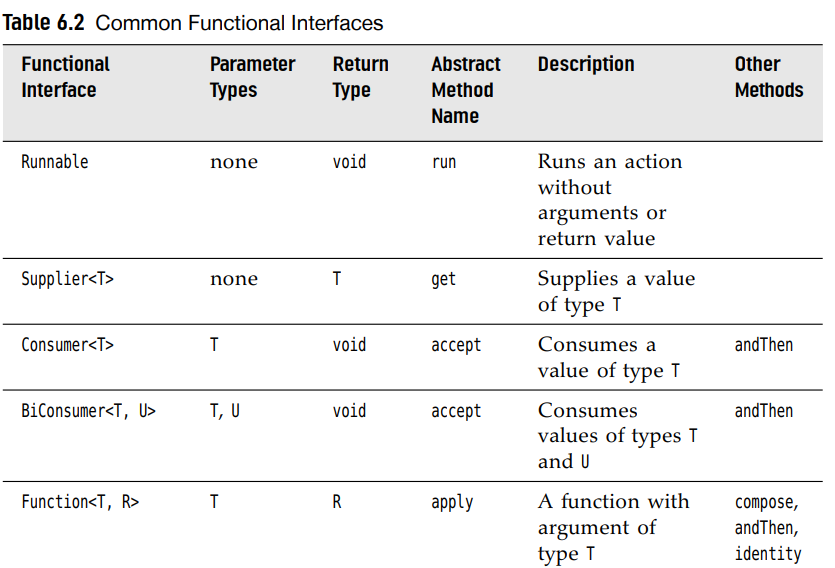


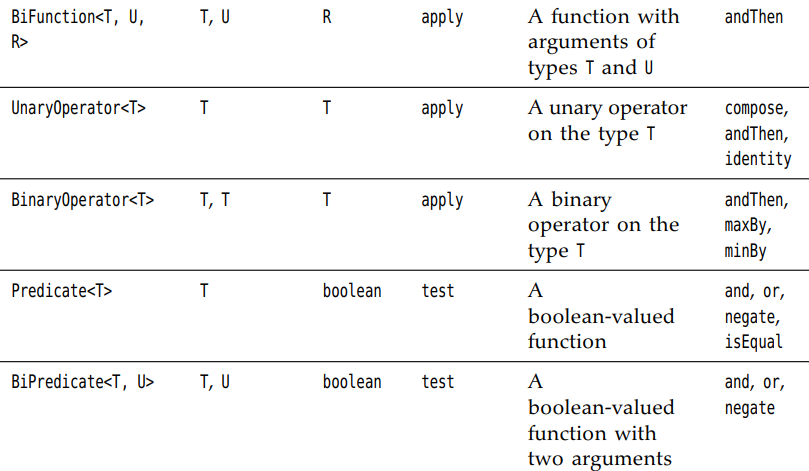
+To accept the lambda, we need to pick a functional interface.

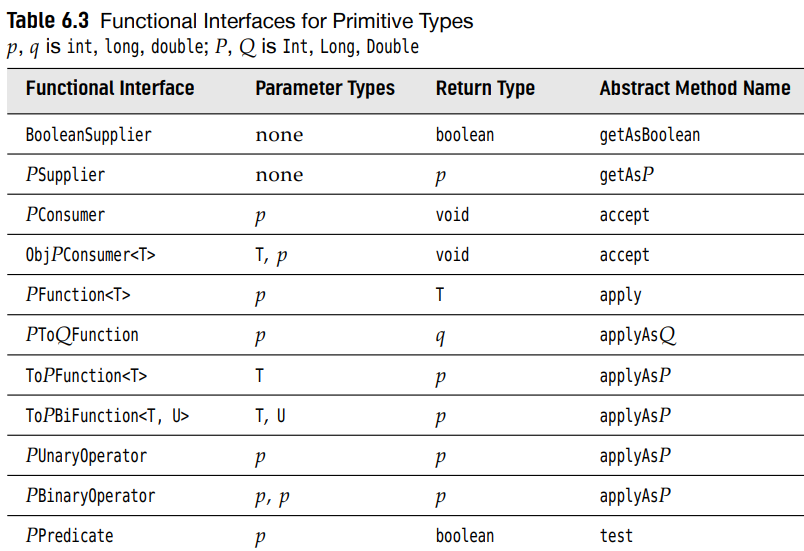


+The body of lambda expression is executed when action.run() is called.

-**Common Functional Interfaces**: p337, 338







-Note: Most of standard functional interfaces have **nonabstract methods** for producing or combining functions. Predicate have 3 default methods: and, or, negate and isEqual().

-**Note**: if you design your own interface with a single abstract method, you can tag it with @FunctionalInterface annotation.

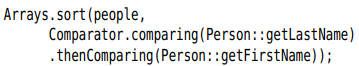
## 6.2.8 More about comparators

-The Comparator interface has a number of static methods for creating comparators. These methods are intended to be used with lambda expressions or method references.

**-comparing**method: take a key extractor function that maps a type T to a comparable type.



-You can chain comparators with **thenComparing**



-There are few variations of these method. You can specify a comparator:



-Both methods have variants that avoid boxing of int, long, double values:



-If your key function can return null, you will like the **nullFirst** and **nullLast** adapters. These methods need a comparator.



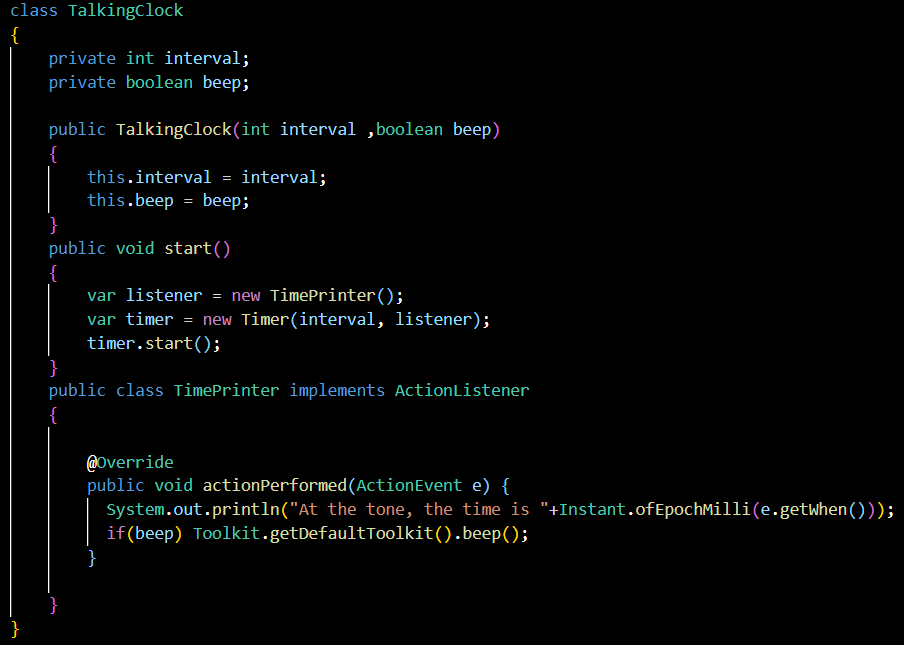
+The **reverseOrder** method

# 6.3 Inner classes

-An **inner class** is a class that is defined inside another class. It can be hidden from other classes in the same package and its methods can access the data from the scope in which they are defined.

-**Inner class** used to implementing **callbacks** (same as lambda expressions) and structuring the code.

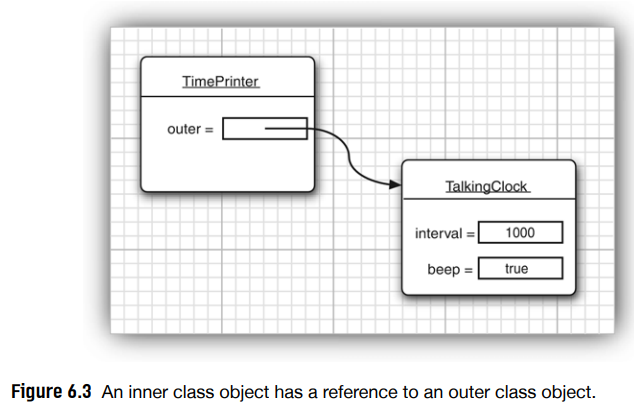
## 6.3.1 Use of an Inner Class to Access Object State



+TimePrinter class is located inside TalkingClock class. This does not mean every TalkingClock has a TimePrinter instance field.

-**Can’t** have **static methods**, **can** have **final static fields**

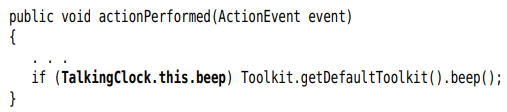
-**Inner class method** gets to **access** both its own **data fields** and **those** of the **outer object** creating it.



## 6.3.2 Special syntax rules for Inner classes

-**OuterClass.this** denotes the **outer class** reference

-**Method**:



-**Inner object constructor** more **explicitly**:







+Refer to **Inner class** when it occurs outside the scope of outer class: 

6.3.3 Are Inner classes useful? Actually necessary? Secure?

-If an inner class accesses a private data field, it can access that data field through other classes added to the package of the outer class

6.3.4 Local inner classes

-You can define the class locally in a **single method**.

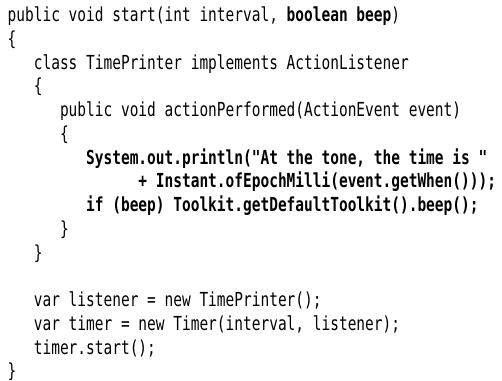


-Local classes are **not declared** with **access specifier**. Their **scope** is restricted to the **block** they’re **declared**

-**Local class** are completely **hidden** from the **outside**

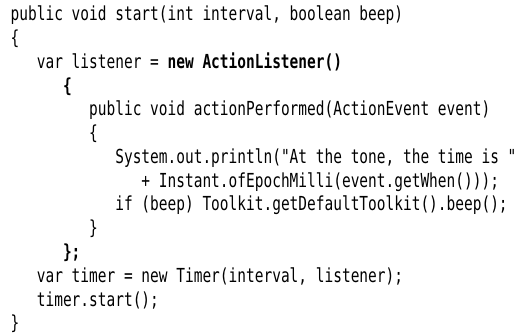
6.3.5 Accessing Variables from Outer Methods

-**Local classes** can even **access** **local variables**. Those variable must be **effectively final***.*

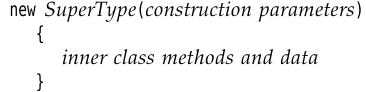


6.3.6 Anonymous Inner Classes

-**Anonymous inner class**: class that don’t have name, used to make a single object of this class



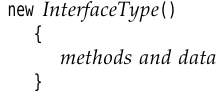
-General syntax:



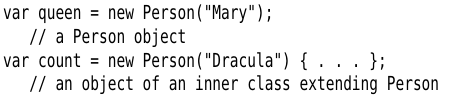
+SuperType can be an **interface, class**

-Anonymous inner class can’t have constructors. The construction parameters are given to superclass constructor.

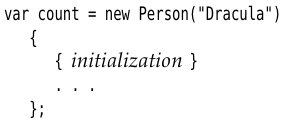
-When an inner class implements interface:



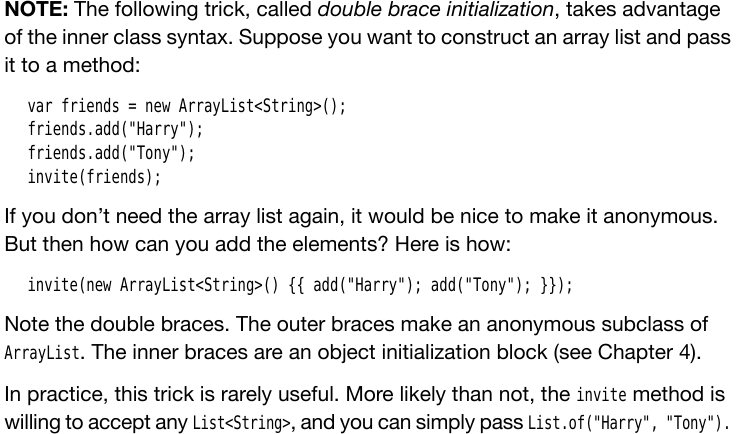
-Different:



-Note: You can provide object initialization block



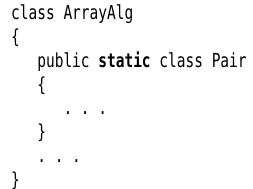
-Note:

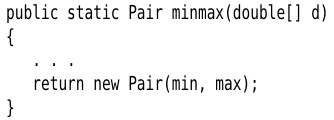


6.3.7 Static inner classes

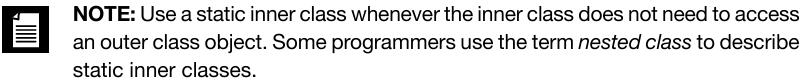
-**static inner class**: exactly like any other inner class, except that an object of a static inner class **doesn’t have** a **reference** to the **outer class object** that generated it.

-Use a static inner class when the inner class object is constructed inside a **static method**.





-Note



# 6.4 Service Loaders

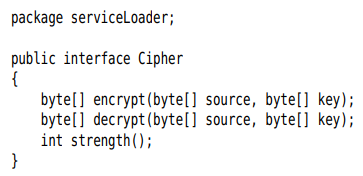
-Sometimes, you develop an app with a service architecture. There are platforms that encourage this approach: OSGi <http://osgi.org/>

-JDK offers a simple mechanism for loading services. This mechanism is supported by Java Platform Module System.

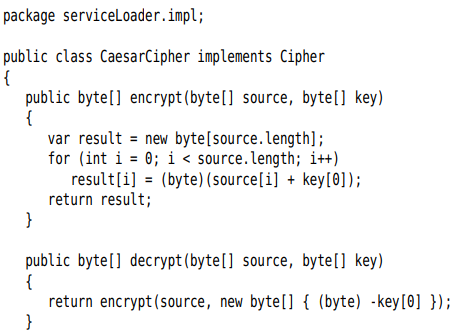
-When providing a service, a program wants to give the service designer some freedom of how to implement the service’s features. The **ServiceLoader** class make it easy to load services that conform to a common interface.

-Define an interface with the methods that each instance of the service should provide

-Example: service provides encryption



+The service provider supplies classes that implement this service:





+Add the names of classes to UTF-8 encoded text file in META-INF/services directory.

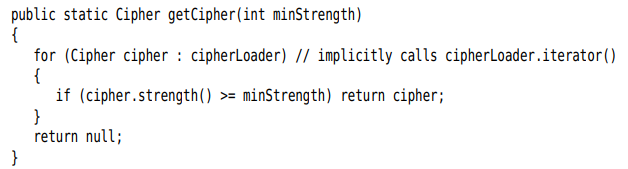
 contains:



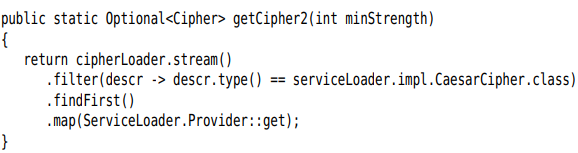
+The program initializes a service loaders:



+The **iterator** method of service loader returns an iterator through all provided implementations of service. It is easiest to use for loop to traverse theme.



+Alternatively, you can use streams to locate the desired service



+To take any service instance:



-API Doc: p362

# 6.5 Proxies

-Skip

6.5.1 When to use proxies

6.5.2 Creating proxy objects

6.5.3 Properties of Proxy classes