Agile Software Development

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

Inheritance in Java

Java Essentials

- **Overview**
  - Introduction
  - Syntax
  - Basics
  - Arrays
- **Classes**
  - Classes Structure
  - Static Members
  - Commonly used Classes
- **Control Statements**
  - Control Statement Types
  - If, else, switch
  - For, while, do-while

- **Inheritance**
  - Class hierarchies
  - Method lookup in Java
  - Use of this and super
  - Constructors and inheritance
  - Abstract classes and methods

  Interfaces
- **Collections**
  - ArrayList
  - HashMap
  - Iterator
  - Vector
  - Enumeration
  - Hashtable

- **Exceptions**
  - Exception types
  - Exception Hierarchy
  - Catching exceptions
  - Throwing exceptions
  - Defining exceptions

  Common exceptions and errors
- **Streams**
  - Stream types
  - Character streams
  - Byte streams
  - Filter streams
  - Object Serialization

Overview

- What is inheritance?
- Implementation Inheritance
  - Method lookup in Java
  - Use of this and super
  - Constructors and inheritance
  - Abstract classes and methods
- Interface Inheritance
  - Definition
  - Implementation
  - Type casting
  - Naming Conventions

What is Inheritance?

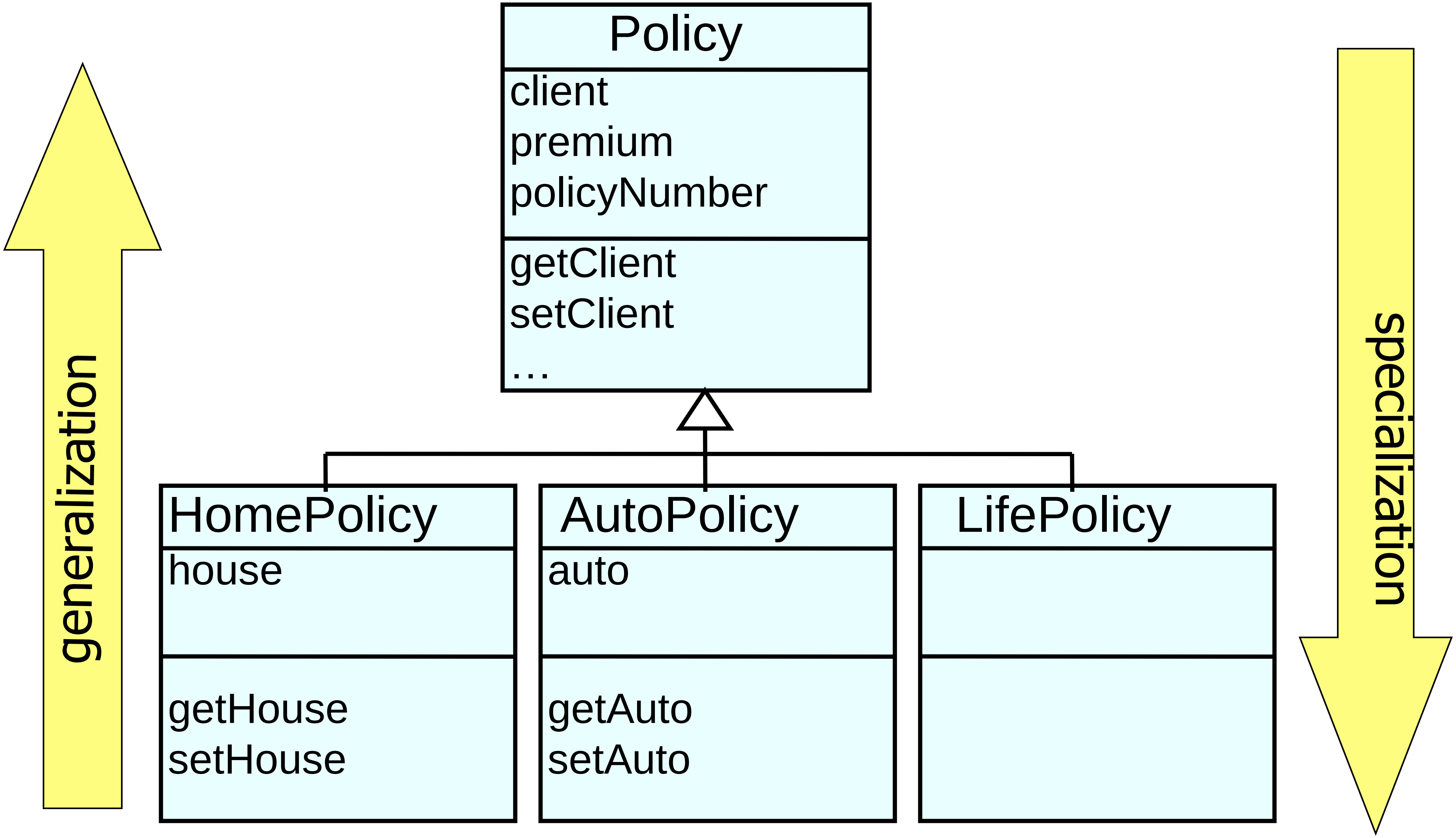
- Inheritance is one of the primary object-oriented principles.
- It is a mechanism for sharing commonalities between classes
- Two types of Inheritance:

1. Implementation Inheritance
   - It promotes reuse
   - Commonalities are stored in a parent class - called the superclass
   - Commonalities are shared between children classes - called the subclasses
2. Interface Inheritance
   - Mechanism for introducing ***Types*** into java design
   - Classes can support more than one interface, i.e. be of more than one ***type***

Overview

- What is inheritance?
- Implementation Inheritance
  - Method lookup in Java
  - Use of this and super
  - Constructors and inheritance
  - Abstract classes and methods
- Interface Inheritance
  - Definition
  - Implementation
  - Type casting
  - Naming Conventions

Implementation Inheritance



Defining Inheritance

- In Java, inheritance is supported by using keyword extends
  - It is said that subclass extends superclass
  - If class definition does not specify explicit superclass, its superclass is Object class

```
public class Policy {…
public class HomePolicy extends Policy{…
public class AutoPolicy extends Policy{…
public class LifePolicy extends Policy{…
```

```
public class Policy{…
```

=

```
public class Policy extends Object{…
```

Variables and Inheritance

- Variables can be declared against the base class, and assigned objects of more derived classes
- E.g. Variable declared as of type Policy can be assigned an instance of any Policy's subclasses

```
Policy policy;
policy = new Policy();
```

```
Policy policy;
policy = new HomePolicy();
```

```
Policy policy;
policy = new AutoPolicy();
```

```
Policy policy;
policy = new LifePolicy();
```

Multiple Inheritance

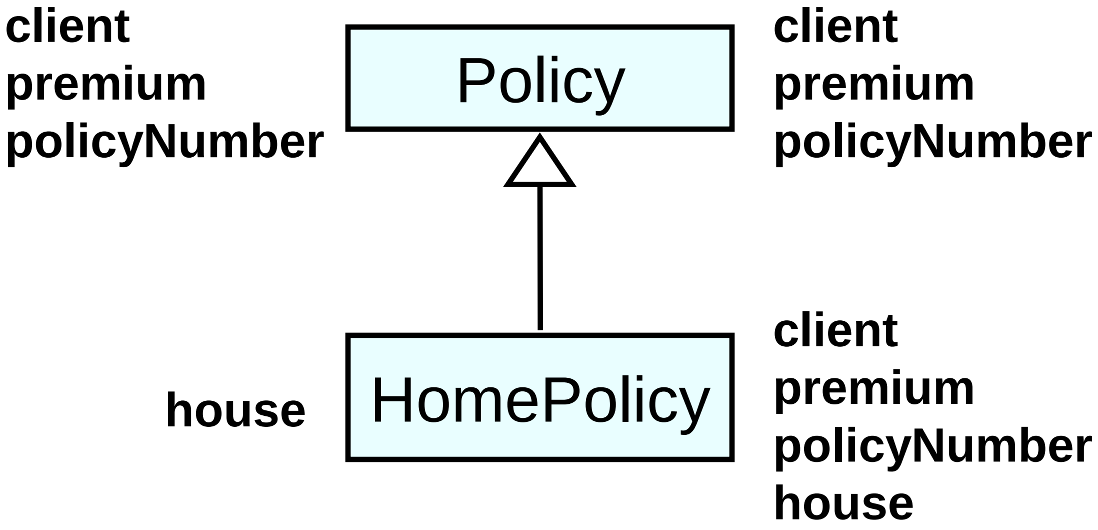
- Not supported in Java
- A class cannot extend more than one class
- There is only one direct superclass for any class
- Object class is exception as it does not have superclass

What is Inherited?

- In general all subclasses inherit from superclass:
  - Data
  - Behavior
- When we map these to Java it means that subclasses inherit:
  - Fields (instance variables)
  - Methods

Inheriting Fields

- All fields from superclasses are inherited by a subclass
- Inheritance goes all the way up the hierarchy

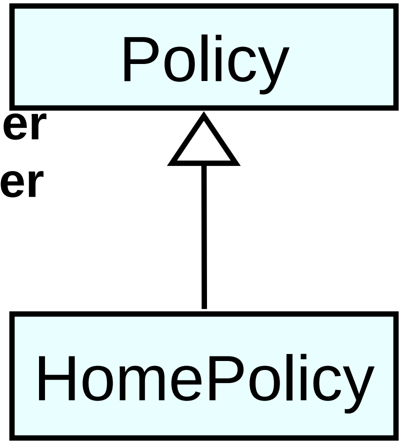


Inheriting Methods

- All methods from superclasses are inherited by a subclass
- Inheritance goes all the way up the hierarchy

getClient
setClient
getPremium
setPremium
getPolicyNumber
setPolicyNumber

getHouse
setHouse



getClient
setClient
getPremium
setPremium
getPolicyNumber
setPolicyNumber

getClient
setClient
getPremium
setPremium
getPolicyNumber
setPolicyNumber
getHouse
setHouse

Method Lookup

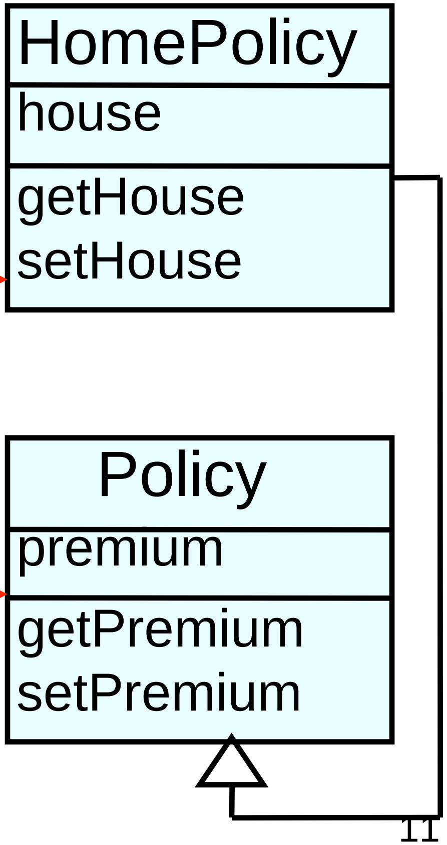
```
…
HomePolicy homePolicy = new HomePolicy();
…
homePolicy.getPremium();
```

- Method lookup begins in the class of that object that receives a message

HomePolicy class – method not found

- If method is not found lookup continues in the superclass

Policy class – method found



HomePolicy

house

getHouse
setHouse

Policy

premium

getPremium
setPremium

this vs. super

- They are both names of the receiver object
- The difference is where the method lookup begins:
  - this
    - Lookup begins in the receiver object’s class
  - super
    - Lookup begins in the superclass of the class where the method is defined

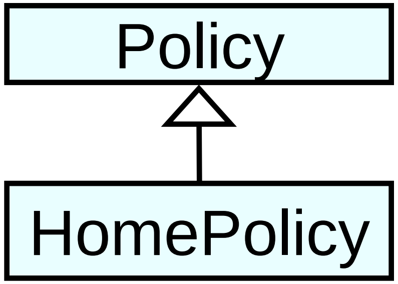
- getClass()
  - Method in java.lang.Object.
  - It returns the runtime class of the receiver object.
- getClass().getName()
  - Method in java.lang.Class.
  - It returns the name of the class or interface of the receiver object.

```
class Policy
{
//…
  public void print()
  {
    System.out.println("A " + getClass().getName() + ", $" + getPremium());
  }
//..
}
```

```
Policy p = new Policy();
p.print();
```

```
A Policy, $1,200.00
```

```
class HomePolicy extends Policy
{
//…
  public void print()
  {
    super.print();
    System.out.println("for house " + getHouse().toString();
  }
//…
}
```

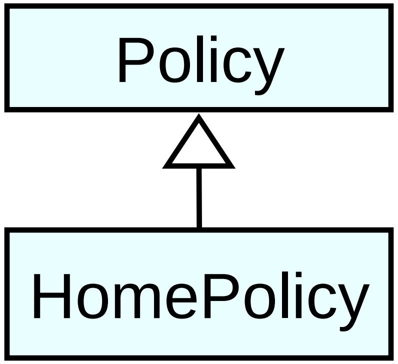


```
HomePolicy h = new HomePolicy();
h.print();
```

```
A HomePolicy, $1,200.00
for house 200 Great Street
```

Method Overriding

- If a class defines the same method as its superclass, it is said that the method is overridden
- Method signatures must match



```
//Method in the Policy class
public void print()
{
  System.out.println("A " + getClass().getName() + ", $" +  getPremium());
}
```

```
//Overridden method in the HomePolicy class
public void print()
{
 super.print();
 System.out.println("for house " + getHouse().toString();
}
```

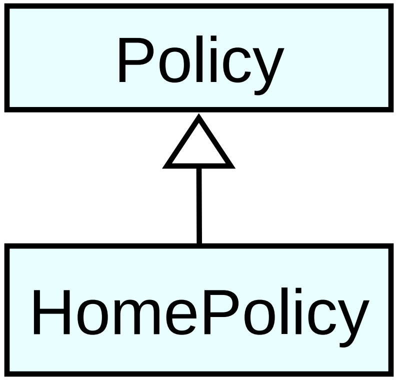
Constructors and Inheritance

- Constructors are not inherited by the subclasses.
- The first line in the subclass constructor must be a call to the superclass constructor.
- If the call is not coded explicitly then an implicit zero-argument super() is called.
- If the superclass does not have a zero-argument constructor, this causes an error.

- Adopting this approach eventually leads to the Object class constructor that creates the object.

Constructors and Inheritance

```
public Policy(double premium, Client aClient, String policyNumber)
{
    this.premium      = premium;
    this.policyNumber = policyNumber;
    this.client       = aClient;
}
```



```
public HomePolicy(double premium,
                  Client aClient,
                  String policyNumber,
                  House  aHouse)
{
    super(premium, aClient, policyNumber);
    this.house = aHouse;
}
```

Abstract Classes

- Classes that cannot have instances
  - They are designed to hold inherited fields and methods for subclasses

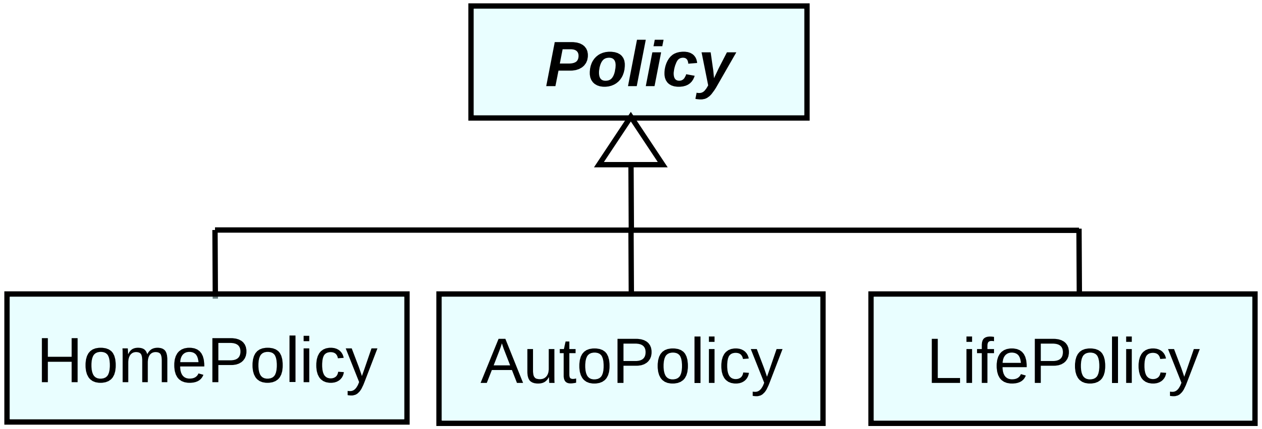
- They also define what subclasses should implement (i.e. through their abstract methods)
  - Details are left for concrete implementation in subclasses

- Usually specified at the design level.

Defining Abstract Classes

- Modifier abstract is used to indicate abstract class

```
public abstract class Policy {…
```



Abstract Methods

- Can only be defined in abstract classes
  - Abstract classes can contain concrete methods as well
  - Declaration of abstract method in concrete class will result in compile error; any class with an abstract method has to be declared abstract.
  - Abstract classes are not required to have abstract methods.

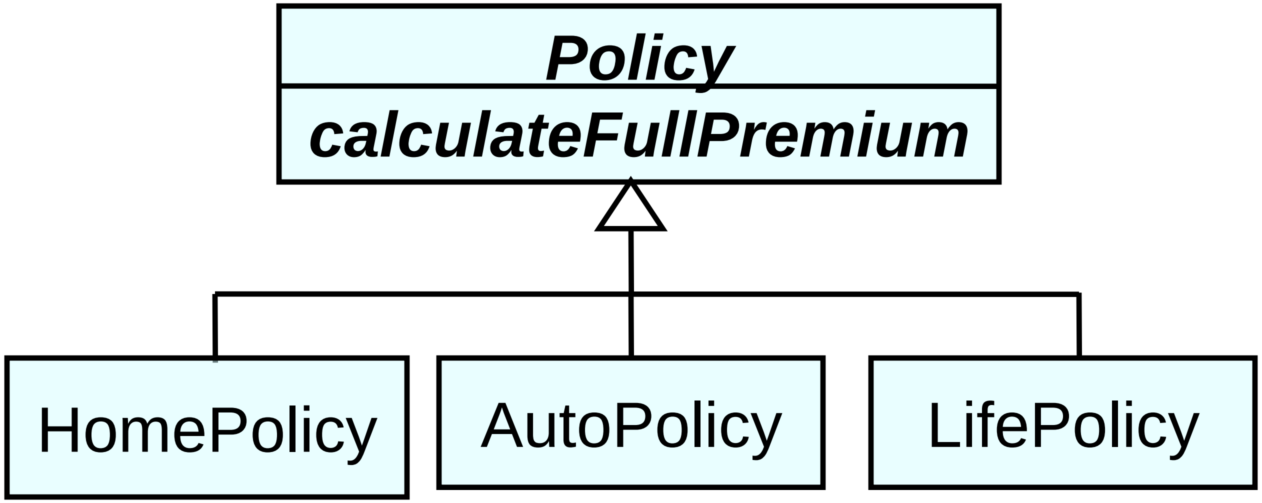
- Declare method signatures
  - Implementation is left to the subclasess
  - Each subclass must have concrete implementation of the abstract method(s)

- Used to impose method implementation on subclasses

Defining Abstract Methods…

- Modifier abstract is also used to indicate abstract method

```
public abstract class Policy
{
  public abstract void calculateFullPremium();
}
```



Policy

calculateFullPremium

HomePolicy

AutoPolicy

LifePolicy

…Defining Abstract Methods

- All subclasses must implement all abstract methods

```
public class HomePolicy extends Policy
{
  //…
  public void calculateFullPremium()
  {
    //calculation may depend on a criteria about the house
  }
}
```

```
public class AutoPolicy extends Policy
{
  //…
  public void calculateFullPremium()
  {
    //calculation may depend on a criteria about the auto
  }
}
```

```
public class LifePolicy extends Policy
{
  //…
  public void calculateFullPremium()
  {
    //calculation may depend on a criteria about the client
  }
}
```

Overview

- What is inheritance?
- Implementation Inheritance
  - Method lookup in Java
  - Use of this and super
  - Constructors and inheritance
  - Abstract classes and methods
- Interface Inheritance
  - Definition
  - Implementation
  - Type casting
  - Naming Conventions

Interface Inheritance

- In Java 8, Interfaces define a set of methods that can be either:
  - abstract: no implementation is provided.
  - default: implementation provided.
  - static: implementation provided.
- Older versions of Java only allowed Interfaces to contain abstract methods.
- Classes that implement interfaces must provide implementation methods for the abstract methods specified in the Interface definition.
- Interfaces are said to specify Types.
- Classes can implement one or more Interfaces as appropriate i.e. have more than one type.

Interfaces Define Types

- Interfaces define Types
  - They define common protocol
  - Can be used to promote design to a higher level of abstraction
  - Can be used where multiple implementations of one abstraction are envisaged
- Interfaces are used to impose typing
  - If a variable is declared as of an Interface type, then an instance of any class that implements that Interface can be assigned to that variable

Defining Interfaces – abstract methods

- Similar to defining classes
  - Keyword interface used instead of class keyword
  - Defined abstract methods contain signatures only (no need for keyword abstract)
  - Interfaces are also stored in .java files
  - Methods are implicitly public access.

```
public interface IAddressBook
{
  void clear();

  IContact getContact(String lastName);

  void addContact(IContact contact);

  int numberOfContacts();

  void removeContact(String lastName);

  String listContacts();
}
```

Defining Interfaces – default methods

- Pre Java 8, adding a new method to an Interface breaks all classes that extend the Interface.
- Java 8 introduced **default methods** as a way to extend Interfaces in a backward compatible way.
- They allow you to add new methods to Interfaces without “breaking” existing implementations of those Interfaces.
- Default method uses the **default** keyword and is implicitly public access.

```
public interface IAddressBook
{
  void clear();

  IContact getContact(String lastName);

  void addContact(IContact contact);

  int numberOfContacts();

  void removeContact(String lastName);

  String listContacts();


  default void typeOfEntity(){
      System.out.println(“Address book”);
  }
}
```

Defining Interfaces – static methods

- In addition to default methods, Java 8 allows you to add **static methods** to Interfaces.

- Use the static keyword at the beginning of the method signature.

- All method declarations in an interface, including static methods, are implicitly public, so you can omit
  the public modifier.

```
public interface IAddressBook
{
  static final int CAPACITY= 1000;

  void clear();

  IContact getContact(String lastName);

  void addContact(IContact contact);

  int numberOfContacts();

  void removeContact(String lastName);

  String listContacts();

  default void typeOfEntity(){
       System.out.println(“Address book”);
  }
  static int getCapacity(){
        return CAPACITY;
  }
}
```

Implementing Interfaces

- Classes implement Interfaces
  - Keyword **implements** is used
  - They must define all abstract methods for the Interface(s) they implement

```
public class AddressBook implements IAddressBook
{
  private Contact[]  contacts;
  private int nmrContacts;

  public AddressBook()
  {
    contacts = new Contact[IAddressBook.getCapacity()];
     nmrContacts = 0;
  }

 private int locateIndex(String lastName)
  {
    //…
  }
  public void clear()
  {
    //…
  }
  //...
}
```

Rules

- Interfaces can contain:
  - Only method signatures for abstract methods
  - Only final static fields
  - default and static methods (including their implementation)

- Interfaces cannot contain:
  - Any fields other than final static fields
  - Any constructors
  - Any concrete methods, other than default and static ones.

Reference vs Interface type

Variable can be declared as:

- Reference type
  - Any instance of that class or any of the subclasses can be assigned to the variable
- Interface type
  - Any instance of any class that implements that interface can be assigned to the variable

```
IAddressBook book;
```

```
book = new AddressBook();
book.clear();
book.addContact(contact);
//… etc…
```

```
book = new AddressBookMap();
book.clear();
book.addContact(contact);
//… etc..
```

book declared as IAddressBook interface type

Variables and Messages

- If a variable is defined as a certain type, only messages defined for that type can be sent to the variable.

```
IAddressBook book;

book = new AddressBook();

book.clear();
book.addContact(contact);

int i = book.locateIndex(“mike”);

// Error – locateIndex() is defined in
// AddressBook – but not in
// IAddressBook
```

Type Casting

- Type casting can be subverted (undermined) by type checking.
- To be used rarely and with care.
- Type cast can fail, and run time error will be generated if the book object really is not an AddressBook
  - (e.g. it could be an AddressBookMap which also implements IAddressBook)

```
IAddressBook book;

book = new AddressBook();

book.clear();
book.addContact(contact);

int i = ((AddressBook)book).locateIndex(“mike”);
```

Type cast from IAddressBook to AddressBook

Interfaces can be Inherited

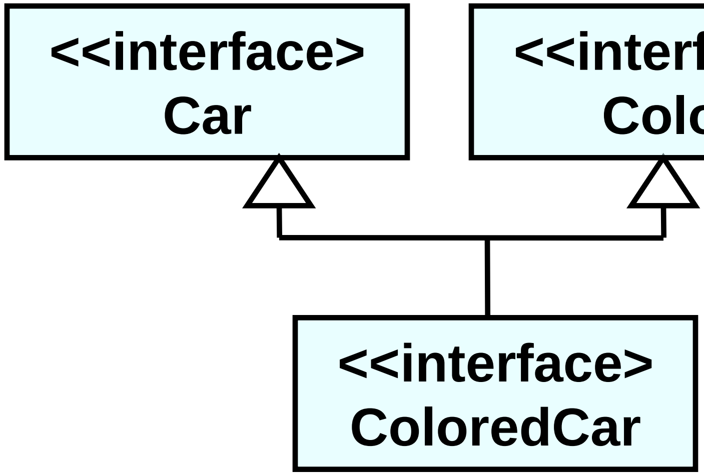
- It is possible that one interface extends other interfaces
  - Sometimes known as “subtyping”
  - Multiple inheritance is allowed with interfaces; whereas a class can extend only one other class, an interface can extend any number of interfaces.

- Inheritance works the same as with classes
  - All methods defined are inherited.

Extending Interfaces

```
public interface Car
{
  public String getSpeed();
}
```

```
public interface Color
{
  public String getBaseColor();
}
```



```
public interface ColoredCar extends Car, Color
{
   public String goFaster();
}
```

Common Naming Conventions

- There are a few conventions when naming interfaces:
  - Suffix **able** is often used for interfaces
    - Cloneable, Serializable, and Transferable
  - Nouns are often used for implementing classes names, and I + noun for interfaces
    - Interfaces: IColor, ICar, and IColoredCar
    - Classes: Color, Car, and ColoredCar
  - Nouns are often used for interfaces names, and noun+Impl for implementing classes
    - Interfaces: Color, Car, and ColoredCar
    - Classes: ColorImpl, CarImpl, and ColoredCarImpl

Review

- What is inheritance?
- Implementation Inheritance
  - Method lookup in Java
  - Use of this and super
  - Constructors and inheritance
  - Abstract classes and methods
- Interface Inheritance
  - Definition
  - Implementation
  - Type casting
  - Naming Conventions