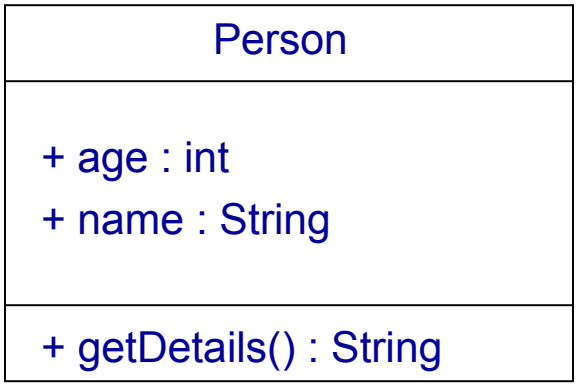
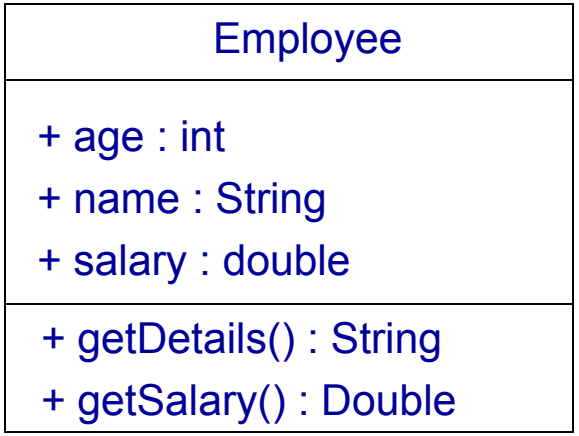
Inheritance

Introduction

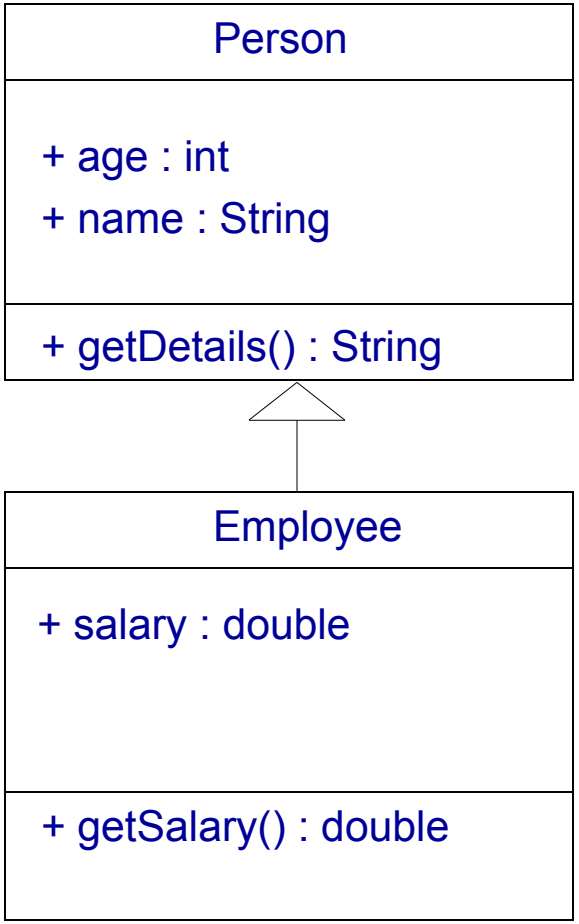
- Sometimes, when creating a class (eg. `Person`) there is a need in a more specialized version of that class (eg. `Employee`).
- `Employee` is a `Person`. A `Person` with additional features.





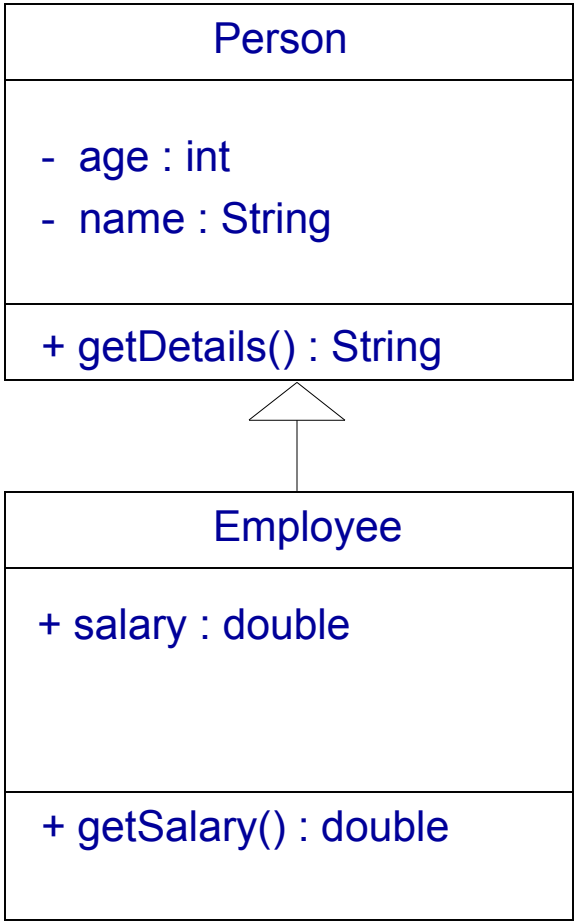
Introduction

- In OOP, inheritance means that one class is defined in terms of a previously defined class. In Java this is achieved using the `extends` keyword.



Introduction

```
class Employee extends Person
{
        public double salary;
        public double getSalary();
}
```



Inheritance in Java

- Java doesn't enable multi-inheritance. A class, in Java, can inherits from only one class. Java supports single inheritance once.

Derivation Syntax

```
[access_modifier] class class_name [extends class_name]
{
    [variables declaration]
    [methods declaration]
}
```

Effects of Inheritance

- Every instance variable that was declared in the parent class will be exist in each one of the objects that were instantiated from the child class.
- Every method that was declared in the parent class – it will be possible to call it on each one of the objects that would be instantiated from the child class.

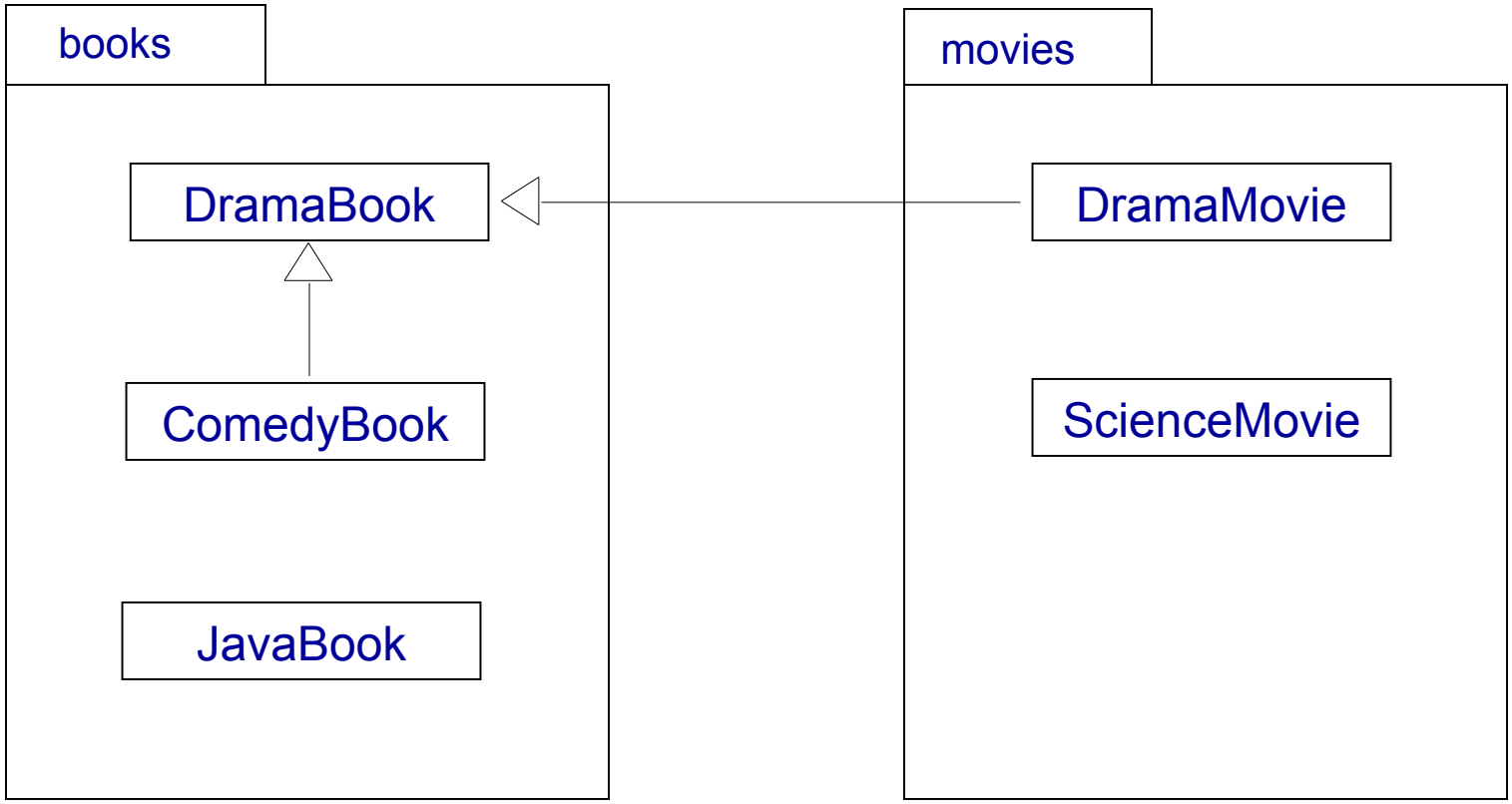
Constructors and Inheritance

- The subclass doesn't inherit the constructors.
- When the sub class is instantiated in addition to the sub class constructor invocation (just before it is executed) the default constructor of the parent class is called.

Access Modifiers

- Each variable and each method can have any of the following access modifiers: private, public, protected and package friendly. The package friendly access modifier is given to method or a variable that their access modifier wasn't specified. It is the default access modifier.

Access Modifiers



Overriding Methods

- Within the sub class we can declare a method with the same name, return type and argument list as those of a method that was inherited from the parent class.
- Doing so, the sub class can modify behavior inherited from the parent class.

Overriding Methods

```
class Person
{
    String name;
    int age;
    String getDetails()
    {
        return
            "name="+name
            +"age="+age;
    }
}
```

```
class Employee extends Person
{
    double salary;
    String getDetails()
    {
        return "name="
            +name +"age="+age
            +"salary="+salary;
    }
}
```

Overriding Methods

```
Person pers = new Employee();
System.out.println(pers.getDetails());
```

The output will be according to the `getDetails()` method defined within the `Employee` class.

Overriding Methods

- All methods in Java are virtual by default.
- The behavior you get is the behavior associated with the runtime type of the object and not the behavior associated with the compiled time type of the variable.

Polymorphism

- polymorphism means many forms. A code that with each one of its execution takes another form is an example for polymorphism.
- The type of the object's reference doesn’t have to be the same as the type of the object itself.

Polymorphism

- Assuming that `Rectangle`, `Triangle` and `Circle` extend `Shape` and given that all of these classes include the area() method we can calculate the total area of all shapes in the following way.

Polymorphism

```
...
Double sum = 0;
Shape vec[] = {new Rectangle(20,30),
               new Triangle(5,4,3),
               new Circle(3)};
for (int index=0; index<vec.length; index++)
{
   sum += vec[index].area();
}
...
```

The `super` Keyword

- When a new object is instantiated from a given class that extends another, the default constructor of the other class is instantiated.
- To invoke another constructor instead we should place a code for doing so. That code should be added as the first line within one of the constructors defined within the child class.

The `super` Keyword

- The call for a specific constructor in the parent class is done using the `super` keyword and it must be the first line within the constructor.
- The specific parent constructor that will be called is the one that fits the arguments we pass over to super.

The `super` Keyword

```
class Employee extends Person
{
   Employee(String str, double sum)
   {
        super(str);
        salary = sum;
   }
}
```

The `super` Keyword

- When overriding a method we can use the super keyword in order to call the version that was overridden.

```
public class Employee extends Person
{
      ...
      public void getDetails()
      {
            ...
            super.getDetails();
      }
}
```

Abstract Class

- Java allows declaring a method that doesn't include an implementation. The implementation of this method is supplied by the sub classes. Such a method is called an abstract method.
- A class with one or more abstract methods is called an abstract class.

Abstract Class

```
public class Test
{
     public static void main(String args[])
     {
             double sum = 0;
             Shape vec[] = {new Circle(3), new Rectangle(4,5),
             new Circle(4), new Circle(8)};
             for(int index = 0; index < vec.length; index ++)
             {
                     sum += vec[index].area();
                     System.out.println(vec[index]);
             }
             System.out.println("Total area is ” + sum);
      }
}
```

Abstract Class

```
abstract class Shape
{
     public abstract double area();
     public String toString()
     {
         return "The area is " + area();
     }
}
```

Abstract Class

```
class Rectangle extends Shape
{
     private double width, height;
     public Rectangle(double wVal, double hVal)
     {
             width = wVal;
             height = hVal;
     }
     public double area()
     {
             return width*height;
     }
}
```

Abstract Class

```
class Circle extends Shape
{
	private double radius;
	public Circle(double rad)
	{
		radius = rad;
	}
	public double area()
	{
		return Math.PI * radius * radius;
	}
}
```

Interfaces

- An Interface is a group of formal declaration of public abstract methods and public final static variables (members).

```
interface <interface_name>
{
     <final & static variables (members) declarations>
     <public abstract methods declarations>
}
```

Interface Default Methods

- As of Java 8 the interface definition can include implemented methods that will be used as the default implementation when a class that implements an interface doesn't implement the methods.

Interface Default Methods

- The default methods provide us with a mechanism for extending interfaces in a backward compatible way.

Interfaces

- The interfaces we define are kind of contracts that specify which methods should be defined in each and every class that implements them.

```
interface IBrowser
{
        public void browse(URL
address);
        public void back();
        public void forward();
        public void refresh();
}

IBrowser browser = new Firefox();
```

Interfaces

- The interface in Java is a reference type, similarly to class.
- The interface can include in its definition only public final static variables (constants), public abstract methods (signatures), public default methods, public static methods, nested types and private methods.

Interfaces

- The default methods are defined with the `default` modifier, and static methods with the `static` keyword. All abstract, default, and static methods in interfaces are implicitly public. We can omit the `public` modifier.
- The constant values defined in an interface are implicitly public, static, and final. We can omit these modifiers.

Interfaces Public Final Static Variables

- Although Java allows us to define enums, many developers prefer to use interfaces that include the definition of public static variables.

```
public interface IRobot {
    public final static int MAX_SPEED = 100;
    public final static int MIN_SPEED = 10;
```

Interfaces Public Abstract Methods

- The public abstract methods we define are actually the interface through which objects interact with each other.

```
public interface IRobot {
    public final static int MAX_SPEED = 100;
    public final static int MIN_SPEED = 10;
    public abstract void moveRight(int steps);
    public abstract void moveLeft(int steps);
    public abstract void moveUp(int steps);
    public abstract void moveDown(int steps);
}
```

Interfaces Sample

```
interface Flyable
{
     public static final int MAX_SPEED=200;
     public abstract void startFlying();
     public abstract void stopFlying();
}
```

Interfaces Sample

- Declaring a class that implements an interface is done using the `implements` key word.

```
class Airplane implements Flyable
{
     ...
}
```

Interfaces

- A class can extend another class and implement a none limited number of interfaces.

```
class Aircraft extends Airplane
      implements Flyable, Printable
{
      ...
}
```

Interfaces

- Using interfaces we can get a common base for objects instantiated from classes that are not related with each other.
- The classes just need to implement the same interface.
- A class can implement many unrelated interfaces.

Interfaces

- Using interfaces it is possible simulating multiple inheritance by declaring a class that implements several interfaces.

Interfaces

```
import java.util.*;

abstract class Shape implements Comparable
{
    abstract double area();
    public int compareTo(Object other)
    {
            return (int)(this.area()-((Shape)other).area());
    }
    public String toString()   { return "area="+area();  }
}
```

Interfaces

```
class Circle extends Shape
{
   double radius;
   Circle(double radiusVal)  { radius = radiusVal; }
   double area()  {  return Math.PI *Math.pow(radius,2); }
   public String toString()
   {
           return "Circle:"+super.toString();
   }
}
```

Interfaces

```
class Rectangle extends Shape
{
    double width, height;
    Rectangle(double widthVal, double heightVal)
    {
            width = widthVal;
            height = heightVal;
    }
    double area()   {  return width*height;  }
    public String toString()
    {
            return "Rectangle:"+super.toString();
    }
}
```

Interfaces

```
class EquilateralTriangle extends Shape
{
    double length;
    EquilateralTriangle(double length)
    {
            this.length = length;
    }
    double area()
    {
            return 0.5*length*Math.sin((2*60*Math.PI/360));
    }
    public String toString()
    {
            return "EquilateralTriangle:"+super.toString();
    }
}
```

Interfaces

```
public class RectangleSortV2
{
    public static void main(String args[])
    {
      Shape vec[] = new Shape[50];
      for(int index=0; index<vec.length; index++)
      {
         switch((int)(4*Math.random()))
         {
           case 1:
             vec[index] = new Rectangle(
                 1000*Math.random(), 1000*Math.random());
             break;
           case 2:
             vec[index] = new Circle(1000*Math.random());
             break;
```

Interfaces

```
                default:
                vec[index] = new EquilateralTriangle(
                    1000*Math.random());
              }
            }
            Arrays.sort(vec);
            for(int index=0; index<vec.length; index++)
            {
              System.out.println(vec[index]);
            }
      }
}
```

Default Methods

- As of Java 8 the interface we define can include the definition of methods together with their implementation.
- This implementation is known as a default one that will take place when the class that implements the interface doesn't include its own specific implementation.

Default Methods

```
public interface IRobot {
    public final static int MAX_SPEED = 100;
    public final static int MIN_SPEED = 10;
    public abstract void moveRight(int steps);
    public abstract void moveLeft(int steps);
    public abstract void moveUp(int steps);
    public abstract void moveDown(int steps);
    public default void turnAround() {
        moveRight(1);
        moveRight(1);
        moveRight(1);
        moveRight(1);

    }
}
```

Default Methods

- When extending an interface that contains a default method, we can let the extended interface inherit the default method, we can redeclare the default method as an abstract method and we can even redefine it with a new implementation.

Interface Default Methods Sample

```
package com.lifemichael.samples;

public interface IPrintable
{
    public abstract void print();
    default public void printStar()
    {
        System.out.print("*");
    }
}
```

Interface Default Methods Sample

```
package com.lifemichael.samples;

public class Circle implements IPrintable
{
    private double radius;
    public Circle(double number)
    {
        setRadius(number);
    }
    public void setRadius(double num)
    {
        if(num>0)
        {
            radius = num;
        }
    }
```

Interface Default Methods Sample

```
    public void printStar()
    {
        System.out.println("x");
    }
    public void print()
    {
        System.out.println(" radius="+radius+" ");
    }
}
```

Interface Default Methods Sample

```
package com.lifemichael.samples;

public class Rectangle implements IPrintable
{
    private double width;
    private double height;
    public Rectangle(double w, double h)
    {
        setWidth(w);
        setHeight(h);
    }
    public void setWidth(double num)
    {
        if(num>0)
        {
            width = num;
        }
    }
```

Interface Default Methods Sample

```
    public void setHeight(double num)
    {
        if(num>0)
        {
            height = num;
        }
    }
    public void print()
    {
        System.out.print(" width="+width+" height="+height+" ");
    }
}
```

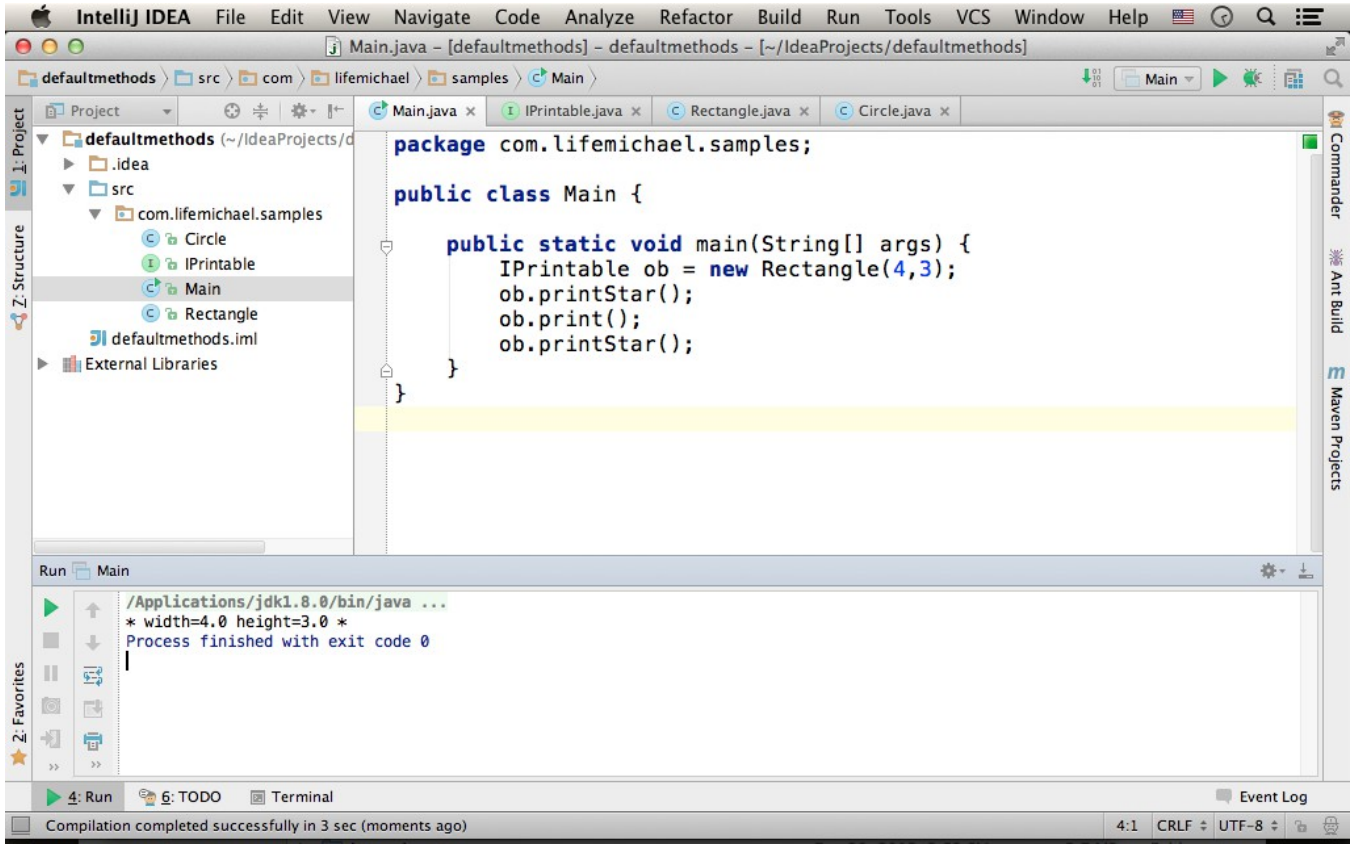
Interface Default Methods Sample

```
package com.lifemichael.samples;

public class Main {

    public static void main(String[] args) {
        IPrintable ob = new Rectangle(4,3);
        ob.printStar();
        ob.print();
        ob.printStar();
    }
}
```

Interface Default Methods Sample



Static Methods

- The interface we define can include the definition of static methods.

Static Methods

```
public interface IRobot {
    public static String getIRobotSpecificationDetails() {
          return "spec...";
    }
    public final static int MAX_SPEED = 100;
    public final static int MIN_SPEED = 10;
    public abstract void moveRight(int steps);
    public abstract void moveLeft(int steps);
    public abstract void moveUp(int steps);
    public abstract void moveDown(int steps);
    public default void turnAround() {
        moveRight(1);
        moveRight(1);
        moveRight(1);
        moveRight(1);

    }
}
```

Nested Types

- The interface we define can include the definition of new static inner types.
- The inner types we define will be implicitly static ones.

Nested Types

```
public interface IRobot {
    public static String getIRobotSpecificationDetails() {
        return "spec...";
    }
    public final static int MAX_SPEED = 100;
    public final static int MIN_SPEED = 10;
    public abstract void moveRight(int steps);
    public abstract void moveLeft(int steps);
    public abstract void moveUp(int steps);
    public abstract void moveDown(int steps);
    public default void turnAround() {
        moveRight(1);
        moveRight(1);
        moveRight(1);
        moveRight(1);

    }
    public static class Engine {
        public void doSomething() {
            getIRobotSpecificationDetails();
        }
    }
}
```

Nested Types

- We can find an interesting example for defining abstract inner type inside an interface when developing a remote service on the android platform.

Nested Types

```
public interface ICurrencyService extends android.os.IInterface
{
    public static abstract class Stub extends android.os.Binder
        implements ICurrencyService
    {

    }

    public double getCurrency(java.lang.String currencyName)
        throws android.os.RemoteException;
}
```

Private Methods

- As of Java 9, the interface we define can include the definition of private methods.
- Highly useful when having more than one default methods that share parts of their implementations. We can place the common parts in separated private methods and make our code shorter.

Private Methods

```
package com.lifemichael;

public interface IRobot {
    ...
    public abstract void moveDown(int steps);
    public default void turnAround() {
        moveRight(1);
        step();
        moveRight(1);
        step();
                ...
    }
    private void step(){
        //...
    }

}
```

Interfaces as APIs

- Companies and organizations that develop software to be used by other developers use the interface as API.
- While the interface is made public, its implementation is kept as a closely guarded secret and it can be revises at a later date as long as it continues to implement the original interface.

Interfaces as Types

- The new defined interface is actually a new type we can use whenever we define a variable or a function parameter.
- Using the interface name as a type adds more flexibility to our code.

Evolving Interfaces

- When we want to add more abstract methods to an interface that was already defined, all classes that implement the old interface will break because they no longer implement the old interface. Developers relying on our interface will protest.

Evolving Interfaces

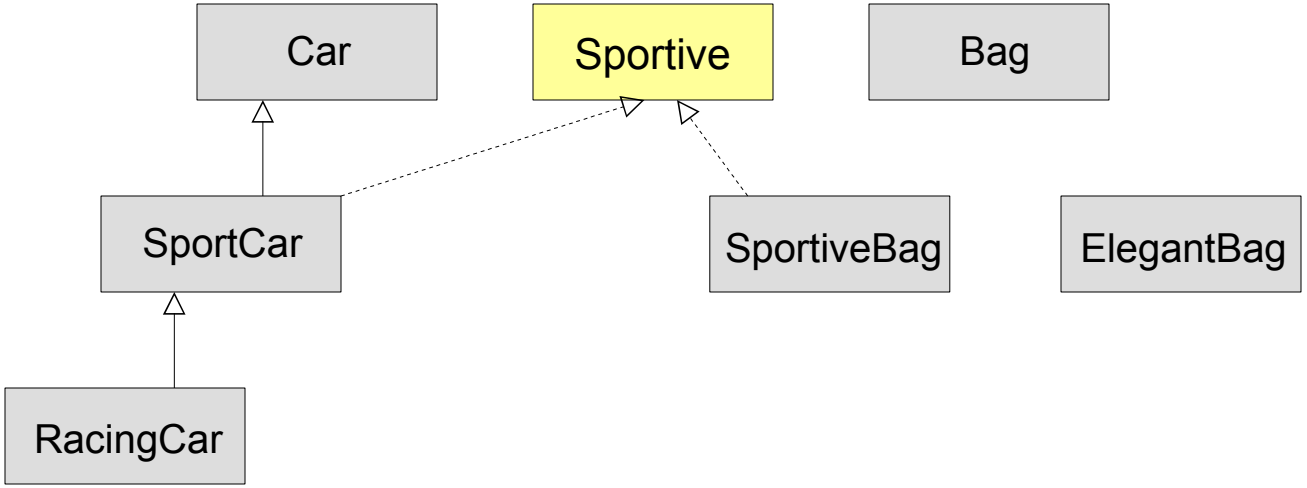
- We can either define a new interface that extend the one we already have, or define the new methods as default ones.
- The default methods allow us to add new functionality to interfaces we already defined, while keeping the binary compatibility with code written for older versions of those interfaces.

Interface as a Dictating Tool

- We can use interfaces for dictating specific requirements from other developers for using functions we developed.

Interfaces as Traits

- Using the default methods we can develop interfaces that will be used as traits in order to reduce the amount of code in our project.



The `instanceof` Operator

- We can use this operator in order to check the type of the object we hold its reference.

```
<reference of object> instanceof <name of class\interface>
```

- The result of the instanceof operator is a boolean value.

  The value is true in each of the following three cases:

  1. The object was instantiated from the class that that its name was specified.

The `instanceof` Operator

2. The object was instantiated from a class that extends the class that was specified.
3. The object was instantiated from a class that implements the interface that was specified.

Casting References

- It is possible casting the type of the reference to another type as long as it can be said that the object itself is also of the type to which the casting is done.
- Using `instanceof` it is possible to perform a safe casting.

Deprecated Methods

- Deprecated methods are 'out of date' methods that you should avoid their usage. These methods were marked in the API Documentation as deprecated.
- As the language evolves new naming patterns are set and new classes are created in order to replace the old ones.

The `Object` Class

- Every class extends `Object` either directly or indirectly.
- Class that doesn't extend another class extends `Object` by default. Therefore, if the class definition doesn't specify a specific base class the `extends Object` code is added.

The `Object` Class

- `Object` has important methods. Each one of these methods can be invoked on each and every object no matter from which class it was instantiated.
- Two of the most important ones are `equals` and `toString`.