

JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

1. Classes and Instances
2. The Methods
3. The Constructors
4. Static Elements
5. Initialization sections
6. Package
7. **Inheritance and Polymorphism**
8. Abstract classes and Interfaces
9. String processing
10. Wrapper classes for primitive types
11. Exceptions and Assertions
12. Nested classes
13. Enums
14. Generics
15. Collections
16. Method overload resolution
17. Multithreads
18. Core Java classes
19. Object Oriented Design
20. Functional Programming

Module contents

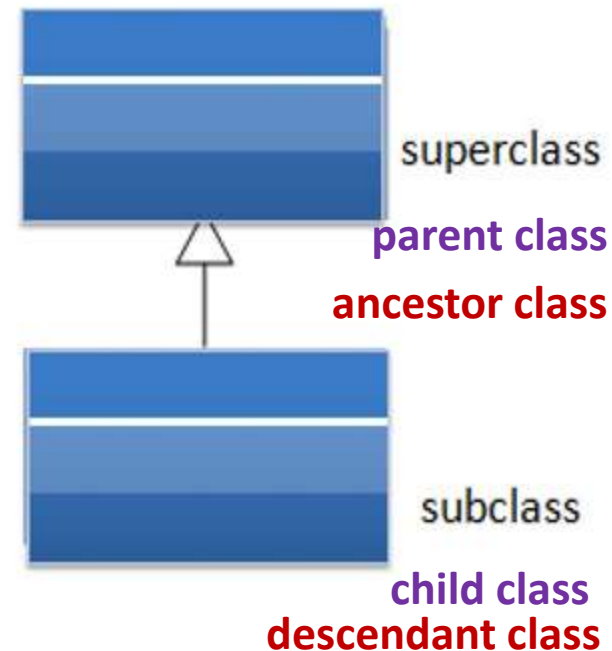
- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - Type cast and conversion
 - The instanceof operator
 - Object cloning
 - Final class and final methods
 - The protected access modifier
 - Sealed classes and interfaces
 - Encapsulating Data with Records

Module contents

- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - Type cast and conversion
 - The instanceof operator
 - Object cloning
 - Final class and final methods
 - The protected access modifier
 - Sealed classes and interfaces
 - Encapsulating Data with Records

The Inheritance 1/5

- Inheritance is the concept of a child class (sub class) automatically inheriting the variables and methods defined in its parent class (super class).
- Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes.



The Inheritance 2/5

- Benefits of Inheritance in OOP : Reusability
- – Once a behavior (method) is defined in a super class, that behavior is automatically inherited by all subclasses
- – Once a set of properties (fields) are defined in a super class, the same set of properties are inherited by all subclasses
- – A subclass only needs to implement the differences between itself and the parent.

The Inheritance 3/5

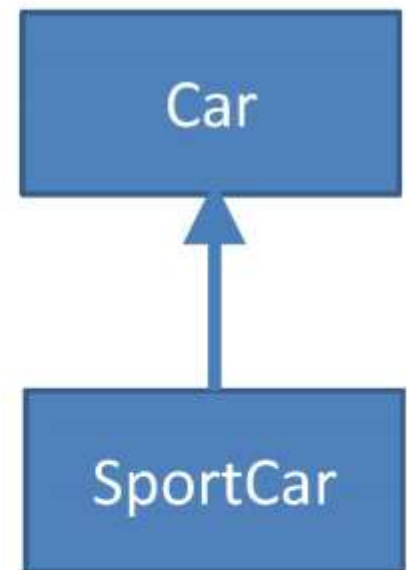
- To derive a child class, we use the **extends** keyword.

```
1. public class Car {  
2.   //...  
3. }
```

} Parent
class

```
4. class SportCar extends Car{  
5.   //...  
6. }
```

} Child
class



The Inheritance 4/5

- A subclass inherits all of the “public” and “protected” members (fields or methods) of its parent, no matter what package the subclass is in
- If the subclass is in the same package as its parent, it also inherits the package-private members (fields or methods) of the parent

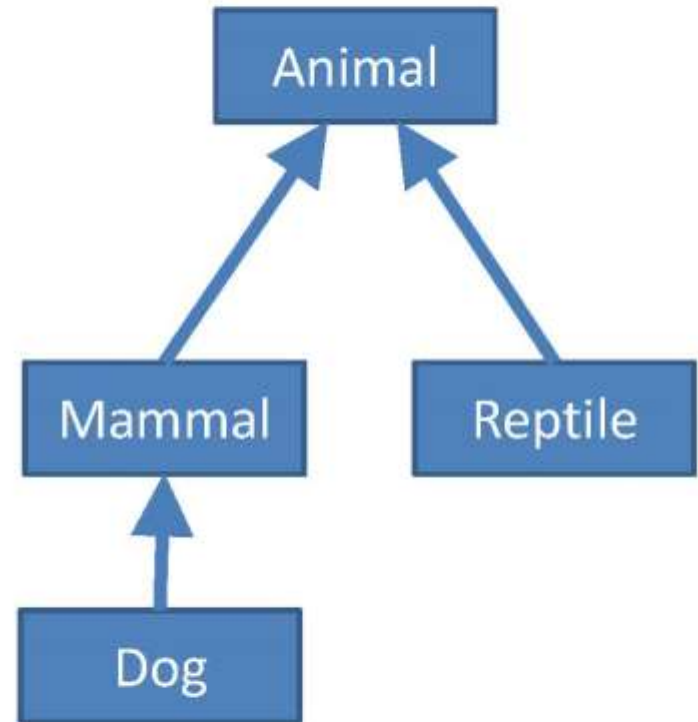
Inheritance and "is-a" relationship 1/2

- IS-A is a way of saying : This object is a type of that object.



Inheritance and "is-a" relationship 2/2

```
1. public class Animal{  
2.     //...  
3. }  
  
4. class Mammal extends Animal{  
5.     //...  
6. }  
  
7. class Reptile extends Animal{  
8.     //...  
9. }  
  
10. class Dog extends Mammal{  
11.     //...  
12. }
```



all subclasses "is-a" superclass

Hierarchy design: common state & behavior - to superclasses

Method overriding

- An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.

For reference types return type
may be superclass type (covariant)

@Override annotation

See BJPoint

Overriding methods can never
be defined as static methods

Fields hiding

- Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different.
- Don't recommend hiding fields as it makes code difficult to read

Instance variable vs instance methods bindings

See fieldshadowpkg

The "super" keyword

- Usage of java super Keyword
- `super()` is used to invoke immediate parent class constructor
- `super` is used to invoke immediate parent class method
- `super` is used to refer immediate parent class instance variable

See BJPoint2

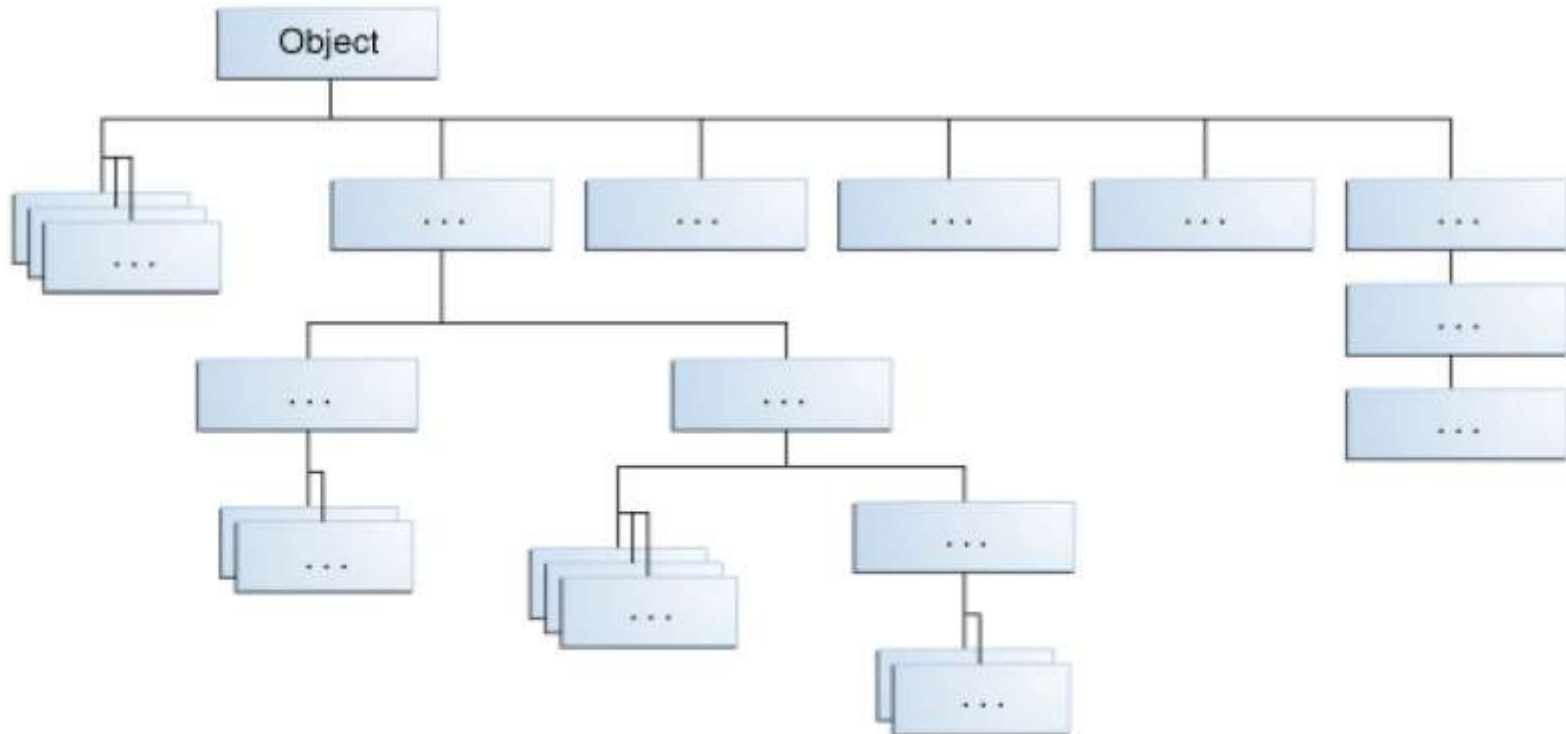
See fieldshadowpkg

Module contents

- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - **Class Object**
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - Type cast and conversion
 - The instanceof operator
 - Object cloning
 - Final class and final methods
 - The protected access modifier
 - Sealed classes and interfaces
 - Encapsulating Data with Records

Class Object 1/3

- At the top of the hierarchy, Object is the most general of all classes `java.lang.Object`



See BJPoint2 - create Point instance and try toString and equals

Object methods: toString(), equals(), hashCode() 1/12

Modifier and Type	Method and Description
<u>protected Object</u>	<u>clone()</u> Creates and returns a copy of this object.
boolean	<u>equals(Object obj)</u> Indicates whether some other object is "equal to" this one.
protected void	<u>finalize()</u> deprecated - use try-with-resources and Cleaners Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
<u>Class<?></u>	<u>getClass()</u> Returns the runtime class of this Object.
int	<u>hashCode()</u> Returns a hash code value for the object.
<u>String</u>	<u>toString()</u> See ObjectMethodsDemo Returns a string representation of the object.

`getClass().getName() + "@" + Integer.toHexString(hashCode())`

Object methods: toString(), equals(), hashCode() 2/12

Modifier and Type	Method and Description
void	<u>notify()</u> Wakes up a single thread that is waiting on this object's monitor.
void	<u>notifyAll()</u> Wakes up all threads that are waiting on this object's monitor.
void	<u>wait()</u> Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	<u>wait(long timeout)</u> Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

Object methods: toString(), equals(), hashCode()... 5/12

- public boolean equals([Object](#) obj)
- Indicates whether some other object is "equal to" this one.
- The equals method for class Object implements returns true if x and y refer to the same object (x == y has the value true).

return (this == obj);

Object methods: toString(), equals(), hashCode() 7/12

- The equals method implements an equivalence relation on non-null object references: **x.equals(null)** -> false
- It is *reflexive* **x.equals(x) -> true**
- It is *symmetric* **x.equals(y) -> true if x.equals(y)**
- It is *transitive* **→ if x.equals(y) -> true & -> true**
- It is *consistent* **y.equals(z) -> true => x.equals(z) -> true**

x.equals(y) ->

true (or false) multiple consistently

See ObjectMethodsDemo

Object methods: toString(), equals(), hashCode() 9/12

- public int hashCode()
- Returns a hash code value for the object
- A **hash function** is any [function](#) that can be used to map digital [data](#) of arbitrary size to digital data of fixed size. The values returned by a hash function are called **hash values**, **hash codes**, **hash sums**, or simply **hashes**.

See ObjectMethodsDemo

Object methods: toString(), equals(), hashCode() 12/12

- If two objects are equal according to the *equals(Object)* method, then calling the *hashCode* method on each of the two objects must produce the same integer result.
- **If you override the *equals()*, you MUST also override *hashCode()*.**

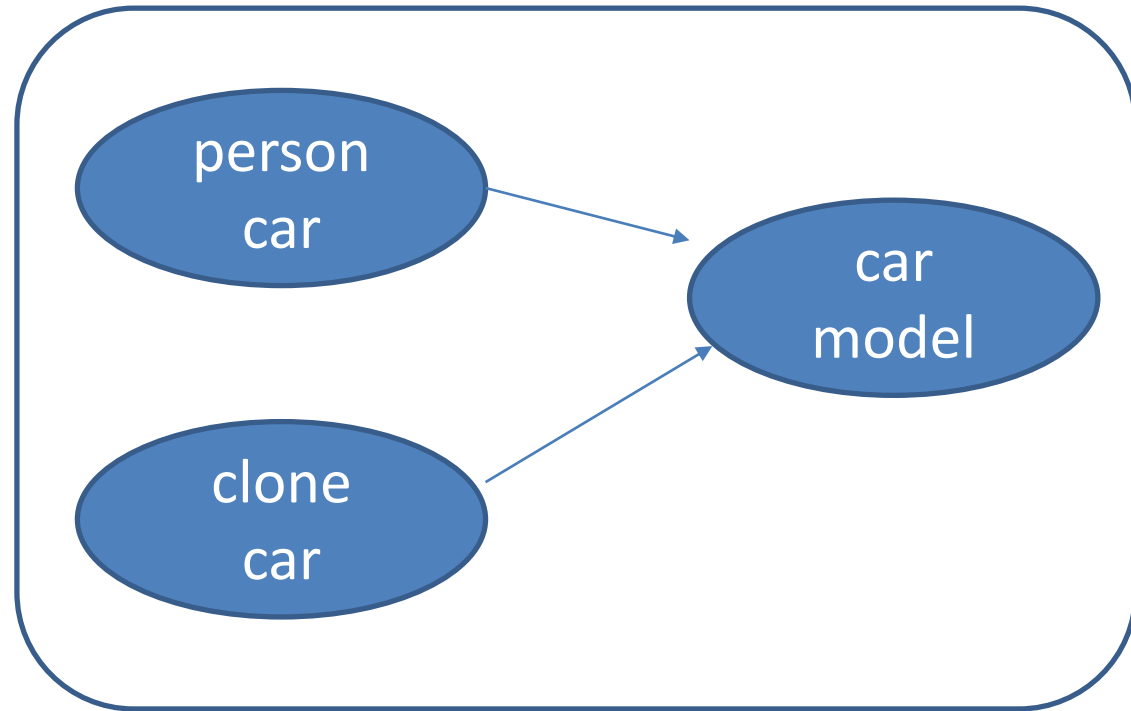
Objects cloning 1/7

- The **object cloning** is a way to create exact copy of an object.
- For this purpose, clone() method of Object class is used to clone an object.

See [objectmetodspkg.clone](#)

Object cloning 2/7

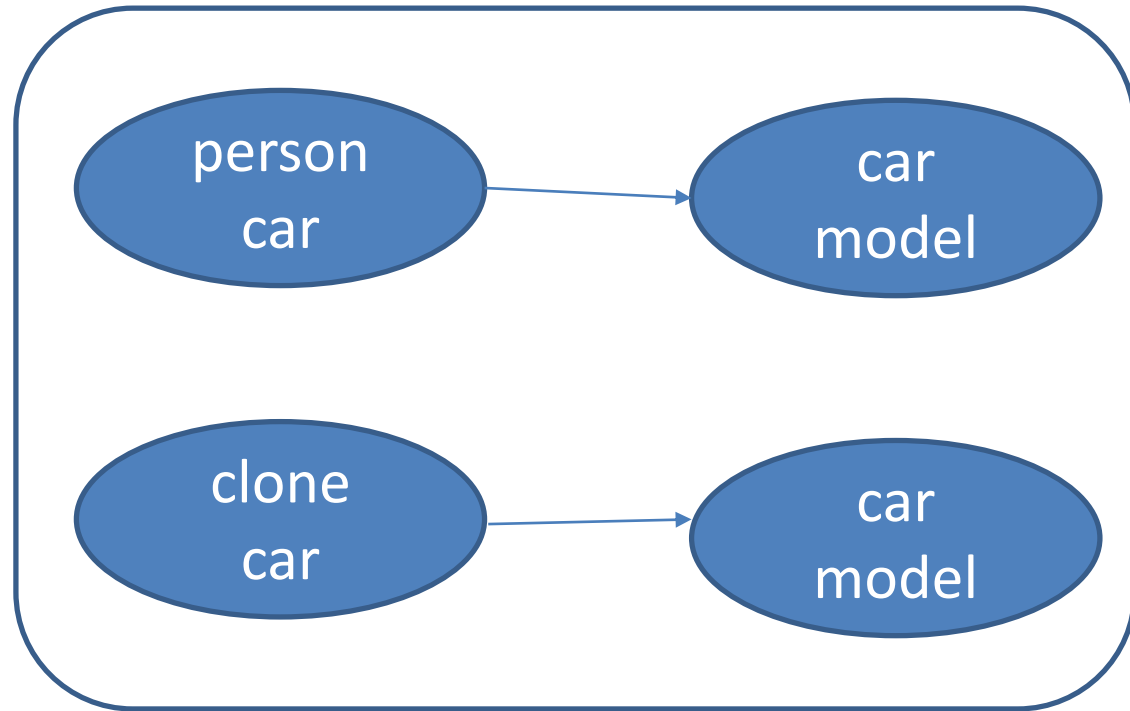
Shallow copy



See `objectmethodspkg.clone`

Object cloning 2/7

Deep copy



See [objectmethodspkg.clone](#)

Module contents

- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - **Constructor chaining**
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - Type cast and conversion
 - The instanceof operator
 - Object cloning
 - Final class and final methods
 - The protected access modifier
 - Sealed classes and interfaces
 - Encapsulating Data with Records

Constructors chaining 1/4

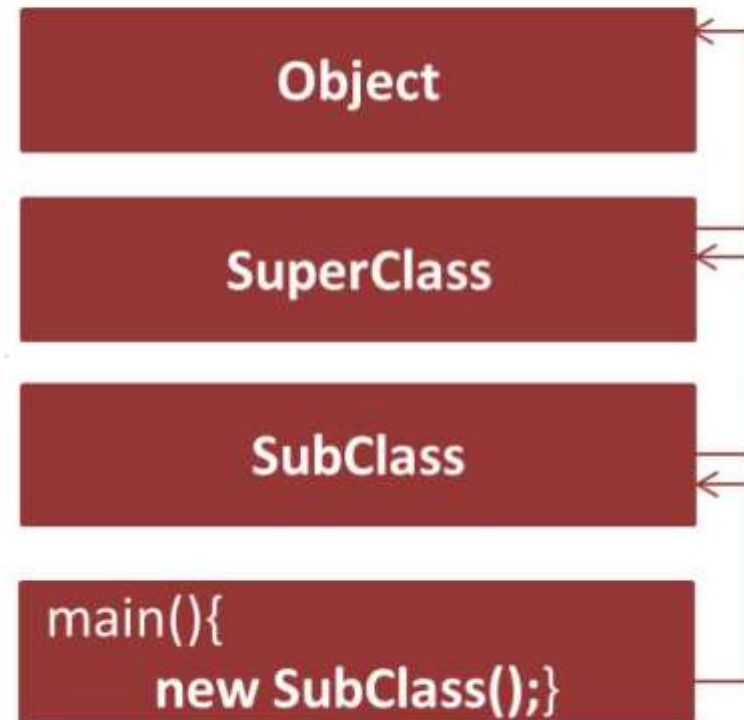
- Calling one constructor from other is called **Constructor chaining in Java**
- Constructors can call each other automatically or explicitly using `this()` and `super()` keywords.
- `this()` denotes a no argument constructor of same class and `super()` denotes a no argument or default constructor of parent class

See `constructorchainingpkg`

Use Force Step Into to see `Object()` invoke

Initialization order and inheritance

```
1. class SuperClass{  
2. }  
3. class SubClass extends SuperClass{  
4. }  
5. public class Start {  
6.     public static void main(String[] arg){  
7.         SubClass c = new SubClass();  
8.     }  
9. }
```



See [construtorchainingpkg](#)

Module contents

- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - **Polymorphism: early binding & late binding**
 - Type cast and conversion
 - The instanceof operator
 - Object cloning
 - Final class and final methods
 - The protected access modifier
 - Sealed classes and interfaces
 - Encapsulating Data with Records

Polymorphism. Early binding & late Binding

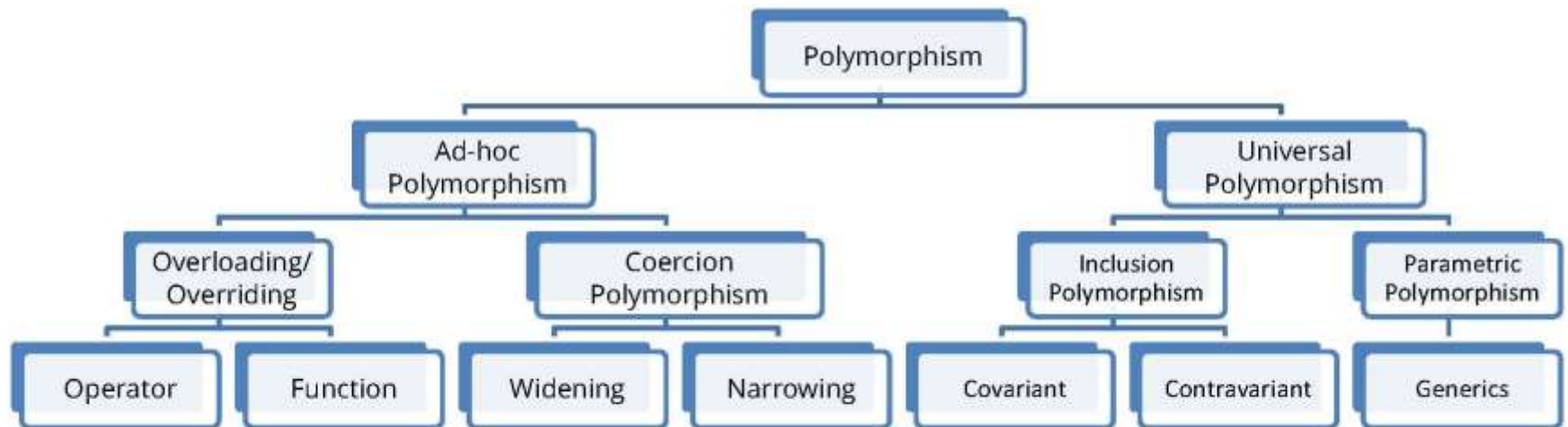
1/5

- **Polymorphism** is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Polymorphism. Early binding & late Binding

2/5

Static Binding or Dynamic Binding



Polymorphism. Early binding & late Binding

3/5

```
1. class Car {  
2.     //...  
3. }  
4. class SportCar extends Car{  
5.     //...  
6. }  
7. class Truck extends Car{  
8.     //...  
9. }  
10. public class Main {  
11.     public static void main(String[] arg) {  
12.         Car myCar = new Car();  
13.         myCar = new SportCar();  
14.         myCar = new Truck();  
15.     }  
16. }
```

See variablevmethodbinding

See variablevmethodbinding.staticmeth

Module contents

- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - **Type cast and conversion**
 - The instanceof operator
 - Object cloning
 - Final class and final methods
 - The protected access modifier
 - Sealed classes and interfaces
 - Encapsulating Data with Records

Type cast and conversion 1/2

- **Java object typecasting** one object reference can be type cast into another object reference. The cast can be to its own class type or to one of its subclass or superclass types or interfaces:
- CastExpression:
`(ReferenceType)RelationalExpression`
- *Example:*
 1. Object obj = **"abcd"**;
 2. String str = (String)obj;
 3. String strBig = str.toUpperCase(); **//OK**

Type casting (downcasting) **expands** the list of methods and properties available for this object!!!

Type cast and conversion 1/2

- Java object conversion (upcasting) - when the object cast to its superclass types or interfaces.

- *Example:*

1. String str = "abcd";

2. Object obj = str;

*/*Cannot resolve method 'toUpperCase' in 'Object'*/*

3. Object objBig = obj.toUpperCase();

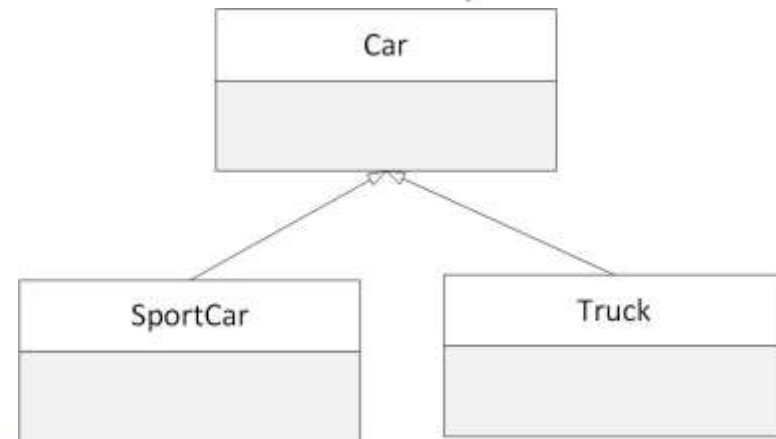
Upcasting **narrows** the list of methods and properties available for this object!!!

Type cast and conversion 2/2

- It is an **ClassCastException** which occurs if you attempt to downcast a class, but in fact the class is not of that type.

```
1. public class Main {  
2.     public static void main(String[] arg) {  
3.         Car myCar = new SportCar();  
4.         SportCar MyCar2 = (SportCar)myCar; // OK!  
5.         Truck MyCar3 = (Truck)myCar; // ClassCastException!  
6.     }  
7. }
```

We have to downcast type
for access to all the object's
members.



Module contents

- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - Type cast and conversion
 - **The instanceof operator**
 - Object cloning
 - Final class and final methods
 - The protected access modifier
 - Sealed classes and interfaces
 - Encapsulating Data with Records

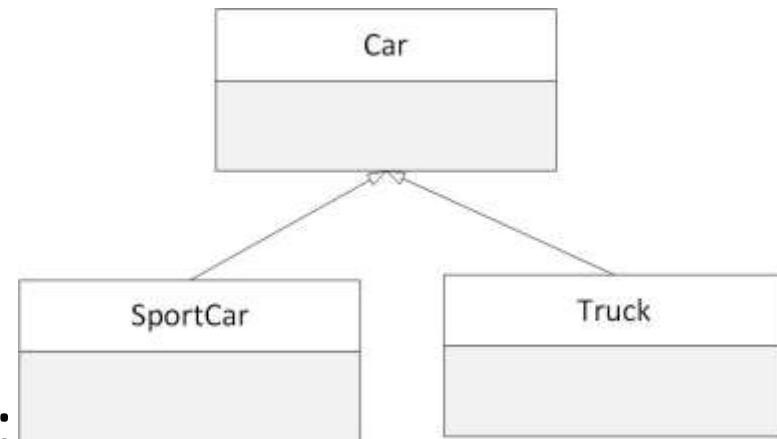
The instanceof keyword 1/2

Object instanceof Reference Type

- At run time, the result of the instanceof operator is true if the *Object* of the *RelationalExpression* is not null and the reference could be cast to the *ReferenceType* without raising a *ClassCastException*. Otherwise the result is false.

- Example:*

```
Car myCar = new SportCar();  
if (myCar instanceof SportCar) {  
    System.out.println("SportCar");  
    SportCar mySportCar = (SportCar) myCar;  
}
```



The instanceof keyword 2/2

- Since JDK 16 - Pattern matching for instanceof
- Since JDK 17(preview) - Enhanced switch with pattern matching

See `patterninstanceof`

See `patternswitch`

See `BJPoint2`

See `point`

Module contents

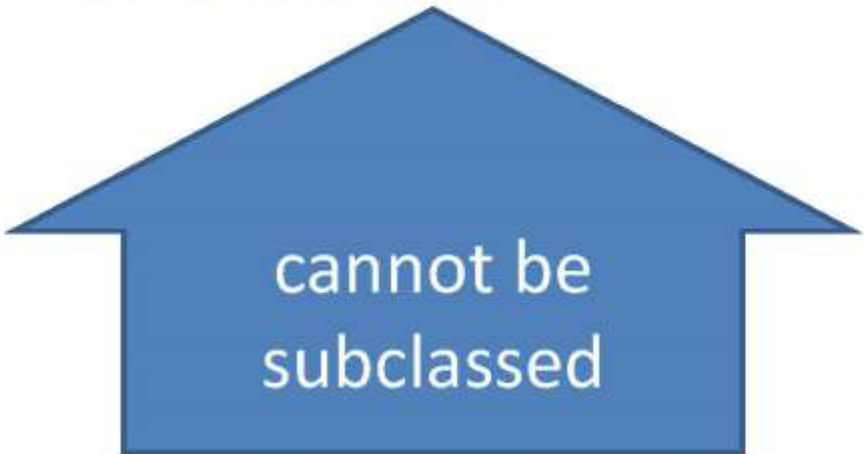
- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - Type cast and conversion
 - The instanceof operator
 - Object cloning
 - **Final class and final methods**
 - The protected access modifier
 - Sealed classes and interfaces
 - Encapsulating Data with Records

Final class and final methods 1/3

- A class that is declared **final** cannot be subclassed.
- **final class** Car { //...
}
- You use the **final** keyword in a method declaration to indicate that the method cannot be overridden by subclasses.
- **public final void** move() { //...
}

Final class and final methods 2/3


```
1. final class Car {  
2.   //...  
3. }  
4. class SportCar extends Car {  
5.   //...  
6. }
```



cannot be
subclassed

Final class and final methods 3/3

- **class** Car {
 //...
 public final void move() {
 System.*out*.println("Car move");
 }
}
final class SportCar **extends** Car {
 //...
 public void move() {
 System.*out*.println("Car move");
 }
}



cannot be
overridden

An interface can't be marked as final

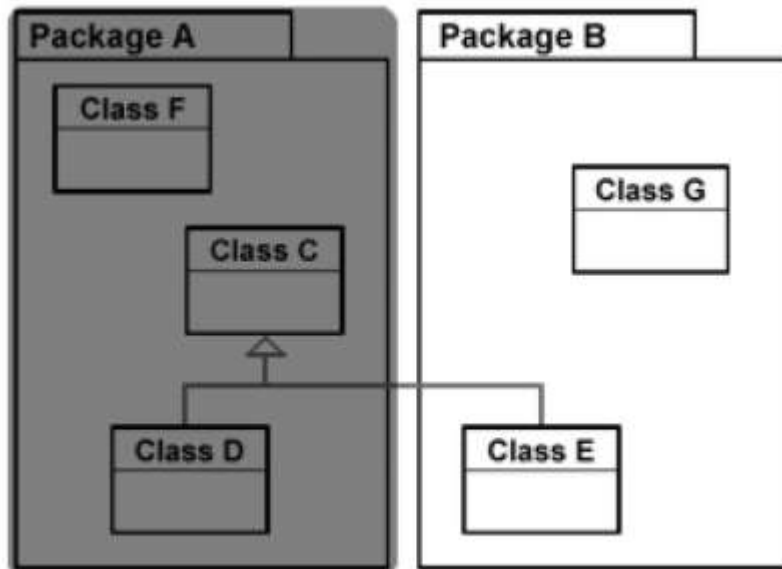
Module contents

- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - Type cast and conversion
 - The instanceof operator
 - Object cloning
 - Final class and final methods
 - **The protected access modifier**
 - Sealed classes and interfaces
 - Encapsulating Data with Records

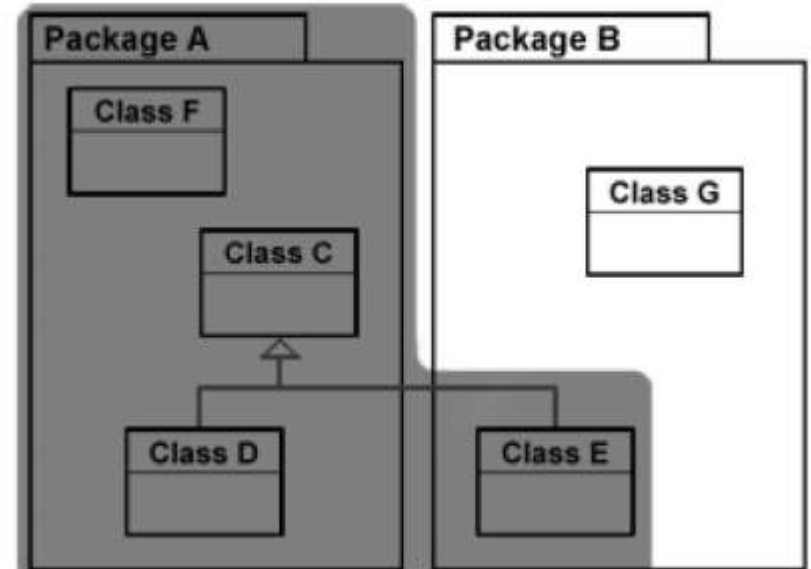
The protected access modifier 1/4

- The **protected** modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

No-modifier (*package-private*)



The **protected** modifier



The protected access modifier 2/4

```
1. package com.brainacad.oop1.mycars;
2. public class Car {
3.     protected int maxSpeed = 180;
4.     //...
5.     protected void move() {
6.         System.out.println(maxSpeed);
7.     }
8. }
```

The protected access modifier 3/4

```
1. package com.brainacad.oop1.spcars2;  
2. import com.brainacad.oop1.mycars.Car;  
3. public class SportCar extends Car {  
4.     //...  
5.     @Override  
6.     public void move() {  
7.         System.out.println( maxSpeed+100);  
8.     }  
9. }
```



The protected access modifier 4/4

```
1. package com.brainacad.oop1.test;
2. import com.brainacad.oop1.mycars.Car;
3. import com.brainacad.oop1.spcars2.SportCar;
4. public class Main {
5.     public static void main(String[] arg){
6.         Car myCar1 = new Car();
7.         SportCar myCar2 = new SportCar();
8.         int s1 = myCar1.maxSpeed; //inaccessible
9.         int s2 = myCar2.maxSpeed; //inaccessible
10.        myCar1.move(); //inaccessible
11.        myCar2.move(); // allowed
12.    }
13. }
```


Module contents

- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - Type cast and conversion
 - The instanceof operator
 - Object cloning
 - Final class and final methods
 - The protected access modifier
 - **Sealed classes and interfaces**
 - Encapsulating Data with Records

Sealed classes and interfaces

- A superclass *with final* modifier **can not** be extended by any classes,
a superclass *without final* modifier **can** be extended by any classes. **Sometimes we need to permit extending a superclass *only some set of classes*.**
- A ***sealed class*** - is a class that restricts which other classes may ***directly*** extend it.
- For sealed class we can limit the ***direct subclasses*** to a ***fixed set of classes***.
- *Example:*
`public sealed class JSONValue`
`permits JSONArray, JSONNumber, JSONString`

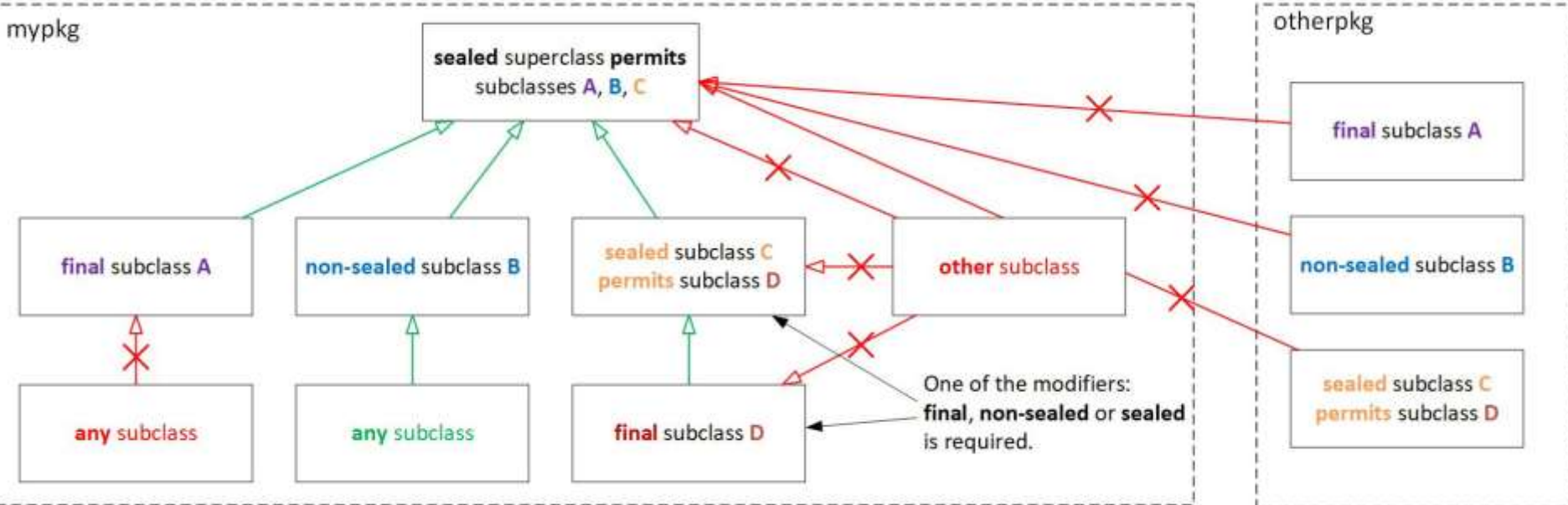
Sealed classes and interfaces

- For sealed interface we can *limit the classes that implement the interface to a fixed set of classes or the interfaces* that extend them.
- Sealed classes must be declared in the **same package** or named module as their direct subclasses.
- The **permits** clause is optional if the sealed class and its direct subclasses are declared within the same file or the subclasses are nested within the sealed class.
- *Example:*

```
public sealed class JSONValue {  
    class JSONArray {...}  
    class JSONNumber {...}  
    class JSONString {...}  
}
```

See sealedclasses

Sealed classes



See sealedclasses

See `sealedclasses.subpkg`

See `sealedclasses.vehicles`

See [sealedclasses.sealedinterface](#)

Module contents

- Inheritance and Polymorphism
 - The inheritance
 - Inheritance and "is-a" relationship
 - Method overriding
 - Fields hiding
 - Class Object
 - Object's methods: toString, equals, hashCode, etc
 - The "super" keyword
 - Covariant return types
 - Constructor chaining
 - Initialization order and inheritance
 - Polymorphism: early binding & late binding
 - Type cast and conversion
 - The instanceof operator
 - Object cloning
 - Final class and final methods
 - The protected access modifier
 - Sealed classes and interfaces
 - Encapsulating Data with Records

Encapsulating Data with Records

- Create final **Student** class with encapsulated final fields: name, surname, age - it length is > 50 LOC of boilerplate code
- Create **StudentR** class-record - it length is 3 LOC
- See **javap -p StudentR**
- Record class is a shallowly immutable, transparent carrier for fixed set of values, called the *record components*
- Record extends **java.lang.Record** abstract class so record can not extend any other class, but can implement interface(s)

record descriptor

public record StudentR(String name, String surname, **int** age) {}

See **java.lang.Record**

since JDK 9

Encapsulating Data with Records

- Create an instance from record
- Create a copy of the instance and check toString, hashCode and equals
- Add unmutator and mutator methods
- Check restriction of canonical constructor by wrong argument
- Add a long constructor with arg restriction
- Replace long constructor by compact constructor
- Add overloading custom constructor
- Implements interface

See records

since JDK 9