Objective

Polymorphism

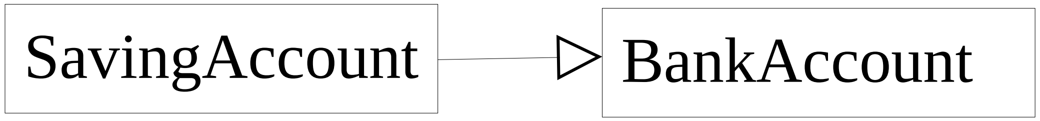
- Abstract class
- Polymorphism
- protected and package access control
- Override cosmetic methods

This lecture covers chapter 9 of your text book.

Review: Inheritance

Inheritance: extend classes by adding methods and fields

Example:



with interest

```
class SavingsAccount extends BankAccount
{
   new methods
   new instance fields
}
```

SavingsAccount automatically inherits all methods and instance fields of BankAccount

Review: Inheritance

Extended class = ***superclass*** (`BankAccount`), extending class = ***subclass*** (`Savings`)

```
SavingsAccount hossein = new SavingsAccount(10);
hossein.deposit(500);
```

Note that we have not implemented deposit(...) in SavingAccount, but we use it since subclass inherits public methods of the super class.

One advantage of inheritance is code reuse

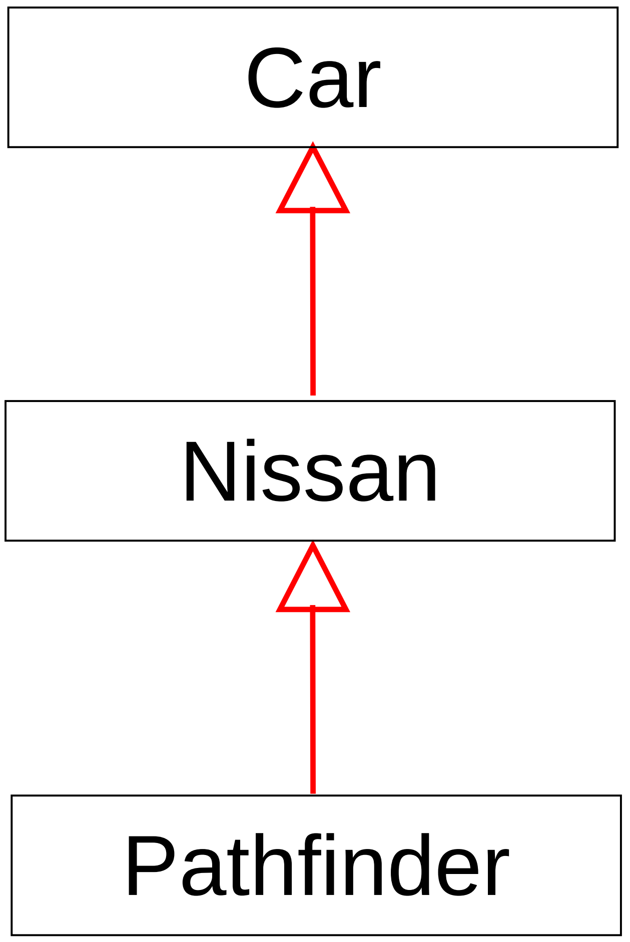
3

An Inheritance Diagram

Inheritance is a “is-a”relation

Pathfinder is a Nissan
Nissan is a Car

but we cannot claim that
Car is a Nissan or
Nissan is a PathFinder



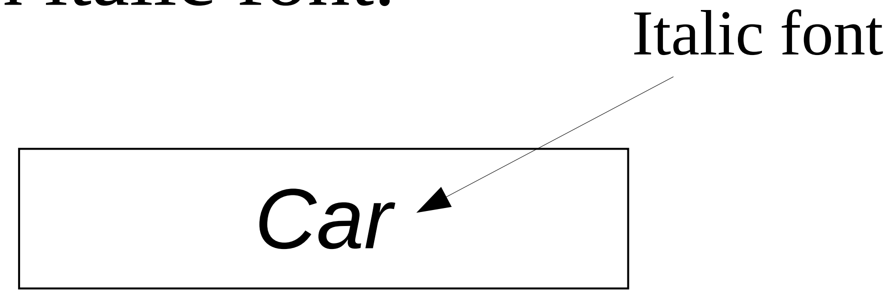
Abstract Classes

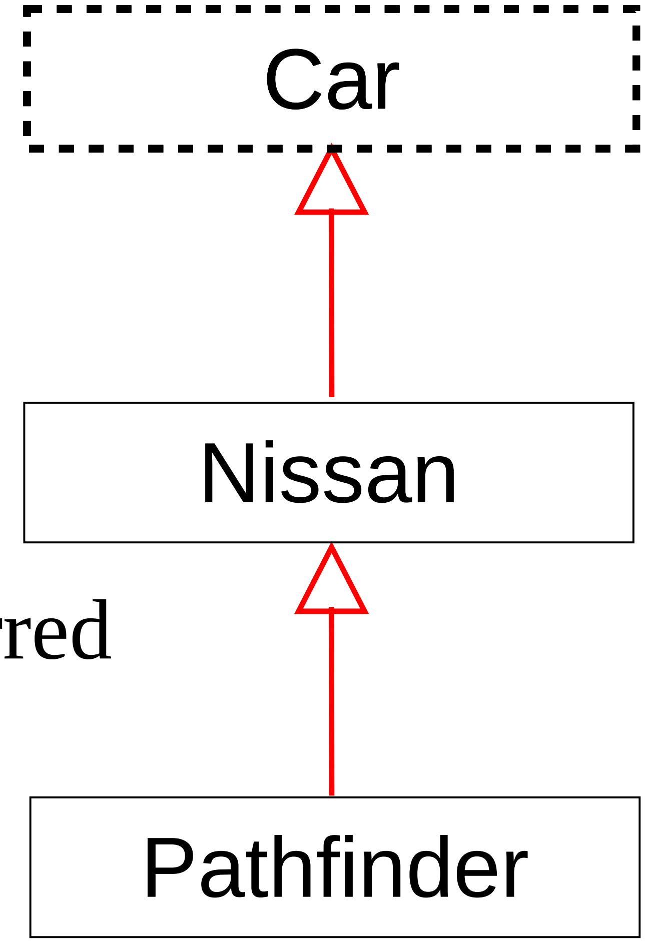
An abstract class in Java is a class that is never instantiated. Its purpose is to be a parent to several related classes to force required functionalities.
For example: Can you imagine a car without brake system?
Are you going to call it a car?

So any car company want to create a new class of car, has to implement brake system, otherwise, they cannot claim that it is a car.

Abstract Classes

In hierarchy drawings abstract classes are drawn with dotted lines
or like other classes but
with italic font.





However, dotted lines is preferred
For clarification

Abstract Classes in Java

```
abstract class ClassName
{
  // Methods and fields like other classes
  // Declaration of abstract methods
}
```

Access modifiers such as public can be placed before abstract.

An abstract class can have regular methods and instance fields like other classes, but they should have at least one abstract method.

Abstract Methods

```
abstract class Vehicle {
  private String type;
  public Vehicle (String type ) {
    This.type = type;
  }
  public String getType() {
    return type;
  }
  public abstract void brake();
  public abstract void accelerator();
   ...
}
```

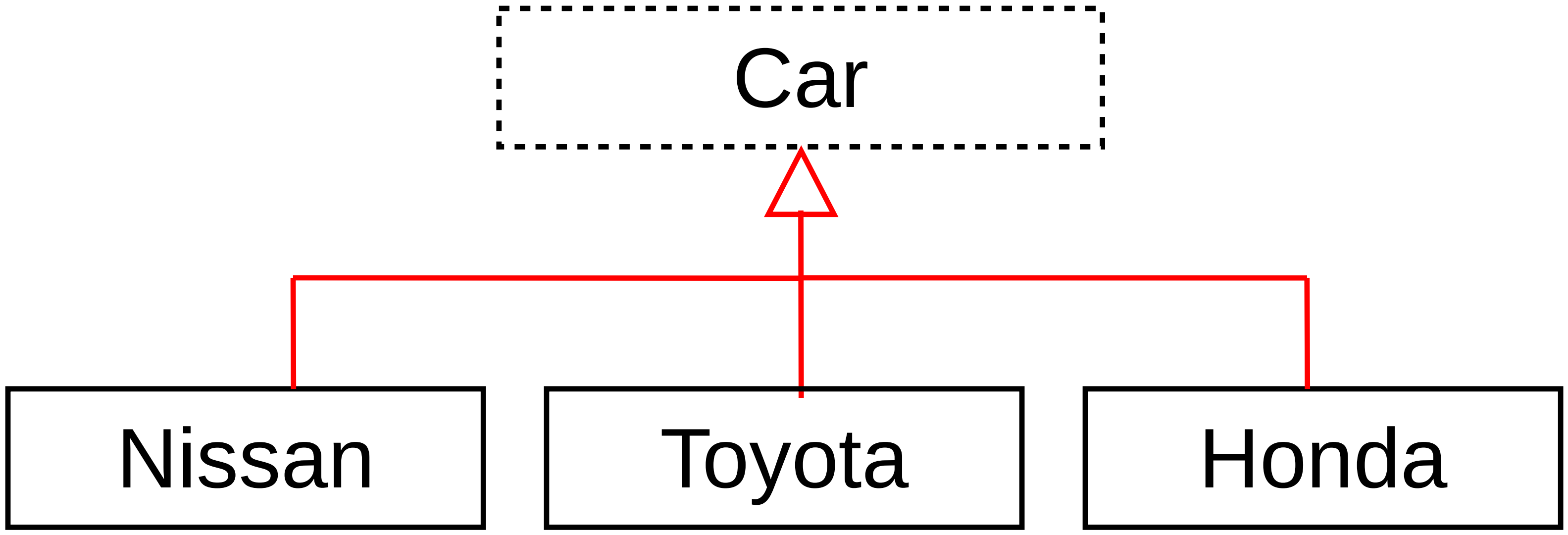
- An abstract method has no body.
- It just declares an *access modifier*, *return type*, and *method signature* followed by a *semicolon*.
- This says that each child inherits the "idea" of `breake(), accelarator(),...`

Example: Abstract Methods

```
abstract class Car {
  private String type;
  public Car ( ) {
    type = "Car";
  }
  public String getType() {
    return type;
  }
  public abstract void printSymbol();
}
```

This says that each child inherits the "idea" of `printSymbol()`

Example: Japanese Cars



In this example, car objects are one of the three types (Toyota, Honda, or Nissan).
Car is used only to group them into a hierarchy.
We cannot create an object of type Car

Abstract Methods

Each of the classes that inherit from **`class Car`** should develop body of printSymbol() method.

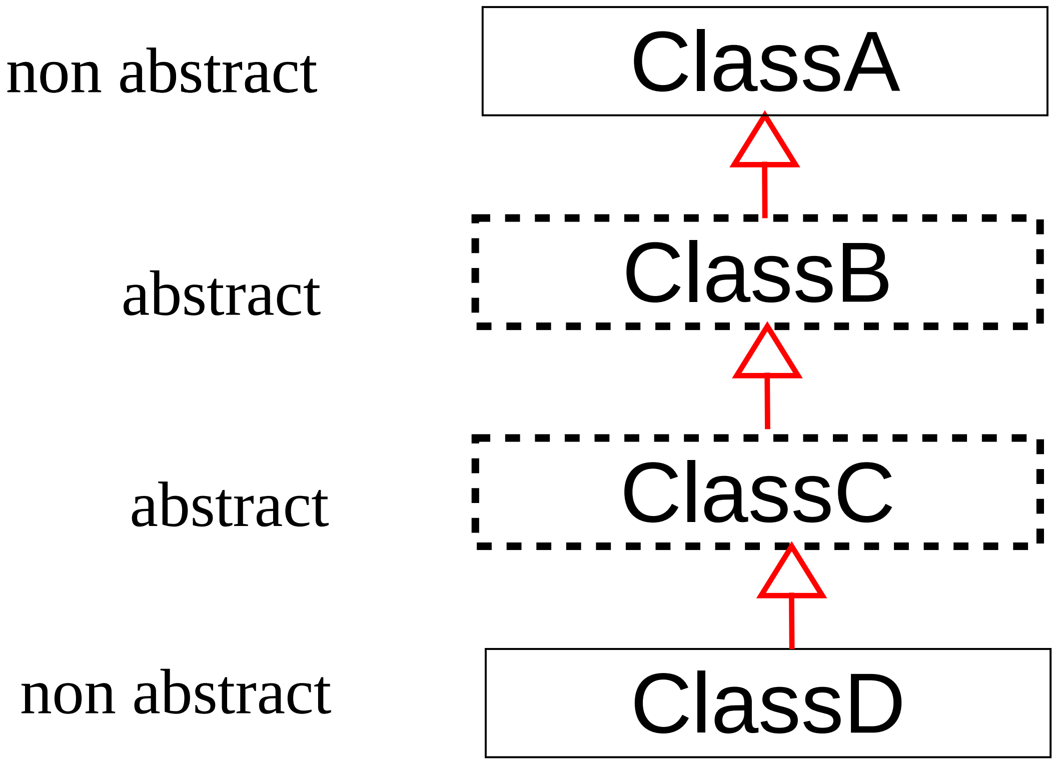
We cannot create an instance of an abstract class. Why?

```
Car aCar = new Car();  // Wrong
```

A non-abstract child class inherits the abstract method, and must define a non-abstract method for each abstract method that matches the signature of the abstract methods.

Abstract Classes

Note that there may be several steps from an abstract base class to a child class that is completely non-abstract.



Example

```
class Honda extends Car {
   public Honda() {
         super();
   }
   // define body of an abstract method
    public void printSymbol()  {
         System.out.println(getType()+" : Honda");
   }
}
```

Car is abstract.

Honda in not abstract anymore.

```
Car myCar = new Car();  // Wrong
Honda myCar = new Honda();  // OK
```

Question

```
Car myCar = new Car();  // Wrong
Honda myCar = new Honda();  // OK

Car aCar = new Honda(); // right or wrong?
```

Question

```
class Benz extends Car {
  private String symbol;
  public Benz( )  {
    super();
    symbol="Benz";
  }
  public String getSymbol(){
    return symbol;
  }
}
```

Right or wrong?

```
Benz myCar = new Benz();
```

What is the problem with this implementation?

Answer

Benz still is an abstract class

```
abstract class Benz extends Car {
  private String symbol;
  public Benz( )  {
    super();
    symbol="Benz";
  }
  public String getSymbol(){
    return symbol;
  }
}
```

Example1

```
public class Example1{
  public static void main(String[] args){
    Honda myCar = new Honda();
    myCar.printSymbol();
  }
}
class Honda extends Car {
  public Honda() {
    super();
  }
  // define body of an abstract method
  public void printSymbol()  {
    System.out.println(getType()+" : Honda");
  }
}
```

Check Example 1

Advantage of Abstract Classes

- Abstract classes are a way of organizing a program.
- You can get the same thing done without using this way to organize. This is a matter of program design, which is not easy at all.
- The advantage of using an abstract class is that you can group several related classes together as siblings. Grouping classes together is important in keeping a program organized and understandable.

Question:

Give another example(s) of advantage of using abstract class:

Example2: `CarTester`

```
class Nissan extends Car {
  public Nissan( )  {
    super();
  }
  // define body of an abstract method
  public void printSymbol()  {
    System.out.println(getType()+" : Nissan ");
  }
}
class Toyota extends Car {
  public Toyota( )  {
      super();
  }
  // define body of an abstract method
  public void printSymbol()  {
    System.out.println(getType()+" : Toyota ");
  }
}
```

Example2: `CarTester` (continue)

```
public class CarTester
{
   public static void main(String[] args)
   {
      Honda myCar1 = new Honda( );
      myCar1.printSymbol();

      Toyota myCar2 = new Toyota( );
      myCar2.printSymbol();

      Nissan myCar3 = new Nissan( );
      myCar3.printSymbol();
   }
}
```

Check Example 2

Output:

```
Car : Honda
Car : Toyota
Car : Nissan
```

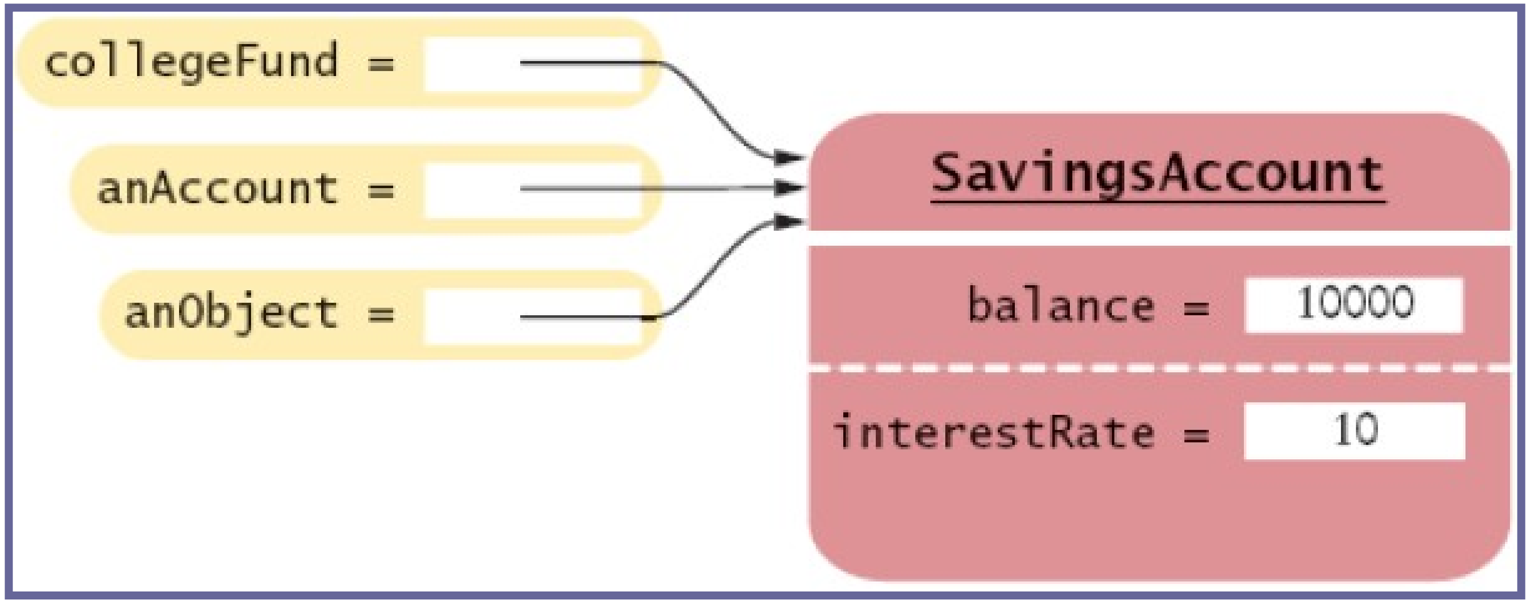
We know that all classes support printSymbol() method.

Review: Reference

Converting Between Subclass and Superclass Types

The three object references all refer to the same object of type SavingsAccount

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
Object anObject = collegeFund;
```



Review: Reference (continued)

All three, collegeFund, anAccount, and anObject refer to the same object of type SavingsAccount.

The collegeFound knows all about the SavingAccount

anAccount only knows part of the story. For example we cannot invoke addInterest(...) method of the object. Since it is no idea about addInterest(...) method.

```
AnAccount.addinterest();  // Wrong
```

AnObject knows only a tiny bit of the story. For example we cannot invoke depost(...) method of the object.

```
AnObject.deposit(100);     // Wrong
```

Review: Reference (continued)

Superclass references don't know the full story:

```
anAccount.deposit(1000); // OK
anAccount.addInterest(); // Wrong
// anAccoount is a BankAccount. It has no idea about
// addInterest() method of SavingAccount
```

When you convert between a subclass object to its superclass type:

- The value of the reference stays the same–it is the memory location of the object
- But, less information is known about the object

Review: Reference (continued)

Superclass references don't know the full story.

anObject has not idea about depost(...) method.

```
anObject.deposit(1000); // Wrong
// anObject is an instance of Object. It has no idea
// about deposit() method of BankAccount
```

Type Casting

Why would anyone want to know *less* about an object?

Answer: re-usability

- Reuse code that knows about the superclass but not the subclass:

```
public void transfer(double amount, BankAccount other)
{
   withdraw(amount);
   other.deposit(amount);
}
```

Can be used to transfer money from any type of `BankAccount`

Polymorphism

Polymorphism means "many forms“

In Java, type of a variable doesn't completely determine type of object to which it refers.

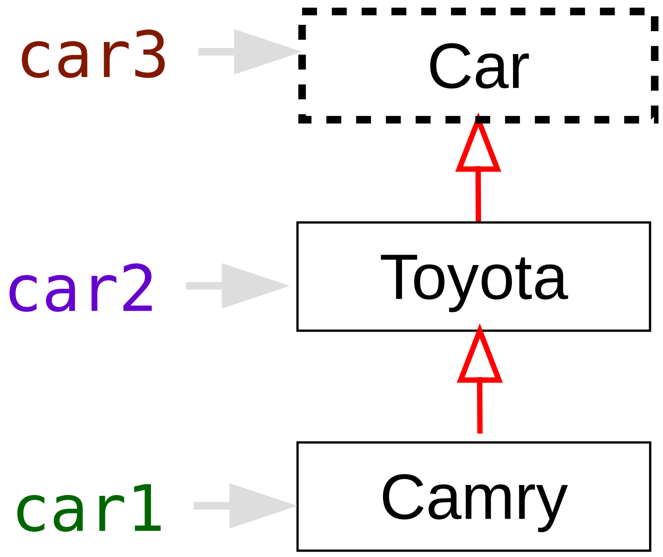
Method calls are determined by type of actual object, not type of object reference.

Example3: Polymorphism

```
class Toyota extends Car {
  public Toyota( )  {
    super();
  }
  // define body of an abstract method
  public void printSymbol()  {
    System.out.println(getType()+" : Toyota ");
  }
}
//----
class Camry extends Toyota{
  public Camry()  {
    super();
  }
  public void printSymbol()  {
    System.out.print("Camry: ");
    super.printSymbol();
  }
}
```

Example3: Polymorphism

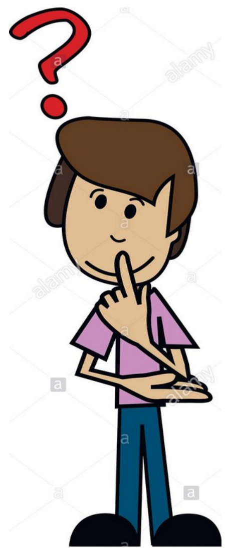
```
Camry car1 = new Camry();
Toyota car2 = car1;
Car car3 = car1;
car1.printSymbol();
car2.printSymbol();
car3.printSymbol();
```



Output:

```
Camry: Car : Toyota
Camry: Car : Toyota
Camry: Car : Toyota
```

Check Example 3



What is going on here?

I assumed that that car2 thinks it is a Toyota, and it has no idea about the Camry, but it knows it! Surprisingly even car3 knows that it is a Camry.

Polymorphism

Note that all classes Car, Toyota, and Camry implemented printSymbol( ) method.

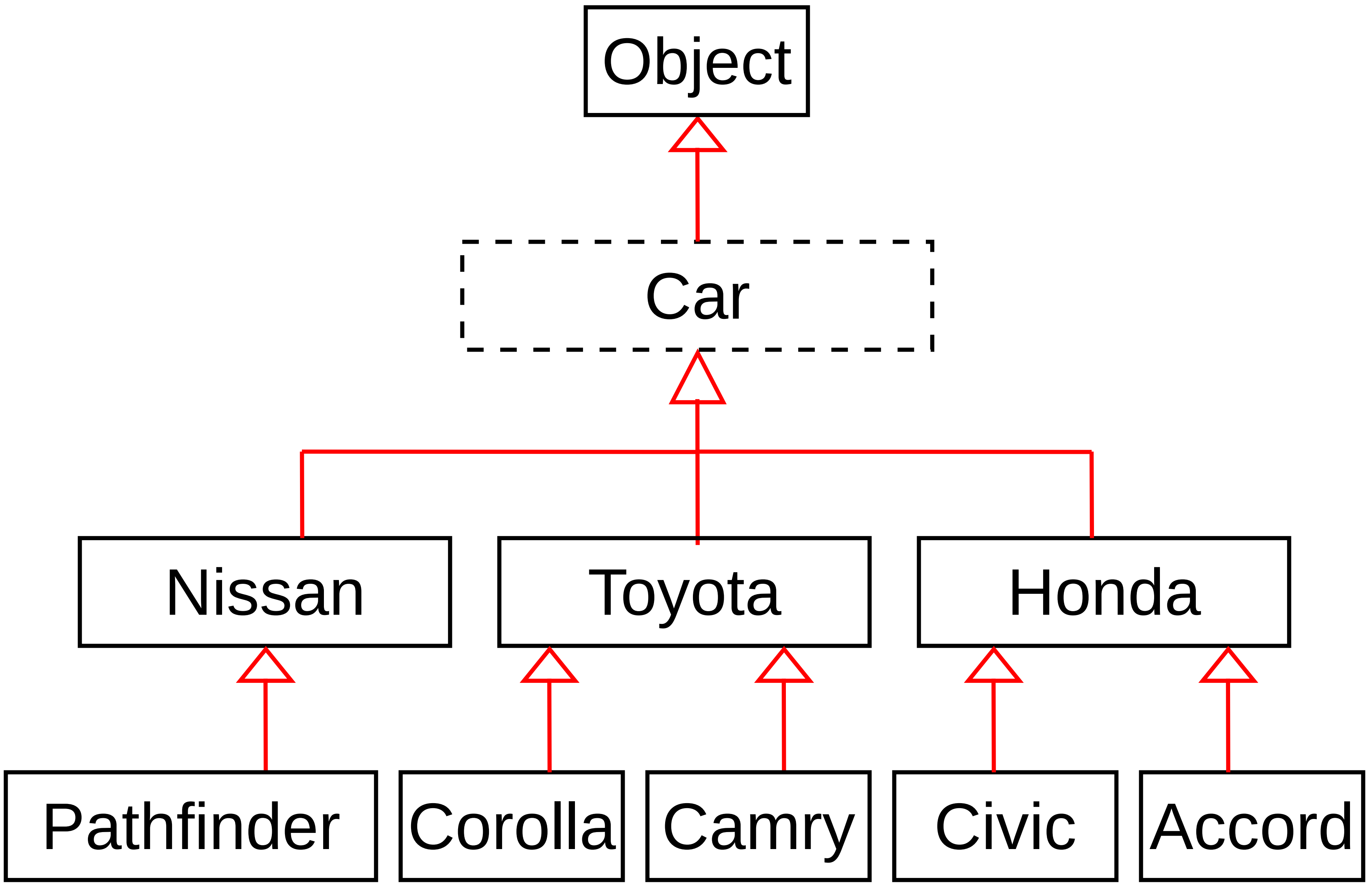
Implementing these methods in class Hierarchy creates an invisible bound, and makes it possible to the object reference to find its own method.

Note that invisible bound mentioned here is just an analogy to understand the problem, not reality.

In Java, type of a variable doesn't completely determine type of object to which it refers.

Method calls are determined by type of actual object, not type of object reference.

Example: JapaneseCars in Java



Polymorphism: JapaneseCarTester.java Example 4

```
class PathFinder extends Nissan{
  public PathFinder()  {
    super();
  }
  public void printSymbol()  {
    System.out.print("Pathfinder: " );
    super.printSymbol();
  }
}
```

Other sup classes are extended the same way.

Refer to JapaneseCarTester.java (example4)

Polymorphism: JapaneseCarTester.java

```
public class JapaneseCarTester{
  public static void main(String[] args) {
    PathFinder myPathF= new PathFinder();
    Civic myCivic = new Civic();
    Accord myAccord = new Accord();
    Honda myHonda;
    Car myCar;
    myHonda =  myCivic;
    myHonda.printSymbol();
    myCar = myAccord;
    myCar.printSymbol();
    myCar = myPathF;
    myCar.printSymbol();
  }
}
```

Check Example 4

Output:

```
Civic: Car : Honda
Accord: Car : Honda
Pathfinder: Car : Nissan
```

Review: Reference

Occasionally you need to convert from a superclass reference to a subclass reference

```
BankAccount anAccount = (BankAccount) anObject;
```

This cast is dangerous: if you are wrong, an exception is thrown.

Solution: use the **`instanceof`** operator

`instanceof`: tests whether an object belongs to a particular type

```
if (anObject instanceof BankAccount) {
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

Polymorphism

Note that compiler only checks if legal methods are invoked

```
Civic myCivic = new Civic();
Honda myHonda;
Object myObj;
myObj = myCivic;  // OK

myObj.printSymbol();   // Wrong

if (myObj instanceof Honda)
{
  myHonda = (Honda)myObj;
  myHonda.printSymbol();
}
```

Example: JapaneseCarArrayTester.java Example 5

```
Object [] objArr = new Object[10];
objArr[0] = new PathFinder();
objArr[1] = "this is a text";
objArr[2] = new Civic();
objArr[3] = new Toyota();
objArr[4] = new Double(10);
objArr[5] = new Camry();
objArr[6] = "another String";
objArr[7] = new Nissan();
for (int i=0; i<objArr.length; i++){
  if (objArr[i] instanceof Car)
    ((Car)objArr[i]).printSymbol();
}
```

Example: Array of Objects

Output:

```
Pathfinder: Car : Nissan
Civic: Car : Honda
Car : Toyota
Camry: Car : Toyota
Car : Nissan
```

Check Example 5

Access Control

Java has four levels of controlling access to fields, methods, and classes:

- public
- private
- protected
- package

Package is default when no access modifier is given. All classes and methods of package access level can be accessed by all classes in the same package. Good opportunity for hackers for critical data.

Recommended Access Levels

Instance and static fields: Always `private`.

Exceptions:

- constants can be public static, since they are safe.
- For non-sensitive data like graphics applications, protected is also acceptable since there are too many parameters, and all subclasses may need to access the super class instance fields

Recommended Access Levels

some objects, such as `System.out`, need to be accessible to all programs (`public`)

occasionally, classes in a package must collaborate very closely, so they can be package.

Methods: `public` or `private`

Recommended Access Levels

Methods: `public` or `private`

Classes and *interfaces*: `public` or package

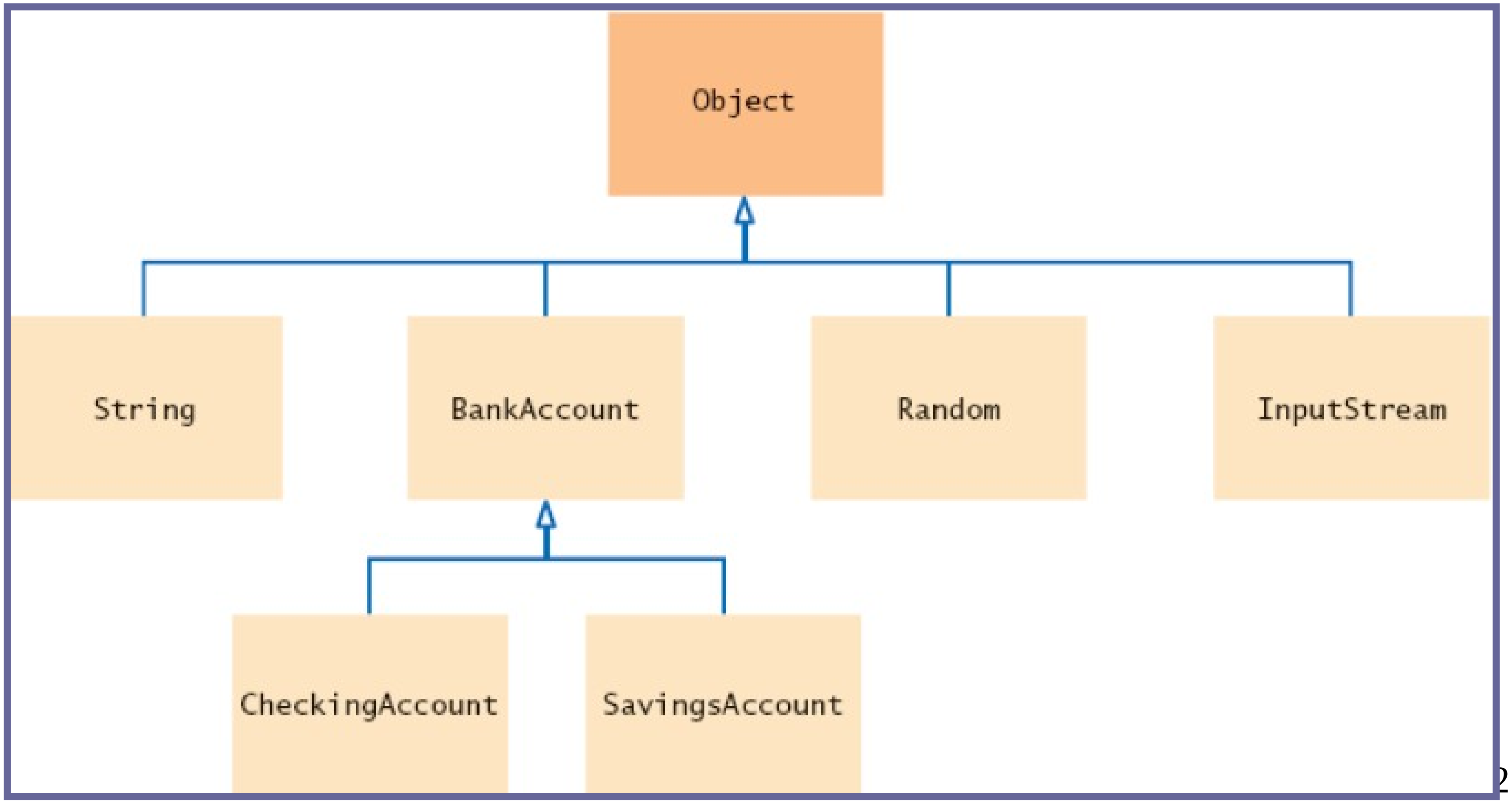
Better alternative to package access: *inner classes (will be covered later)*

In general, *inner classes* should not be `public` (some exceptions exist, `Ellipse2D.Double`)

Beware of accidental package access (forgetting `public` or `private`)

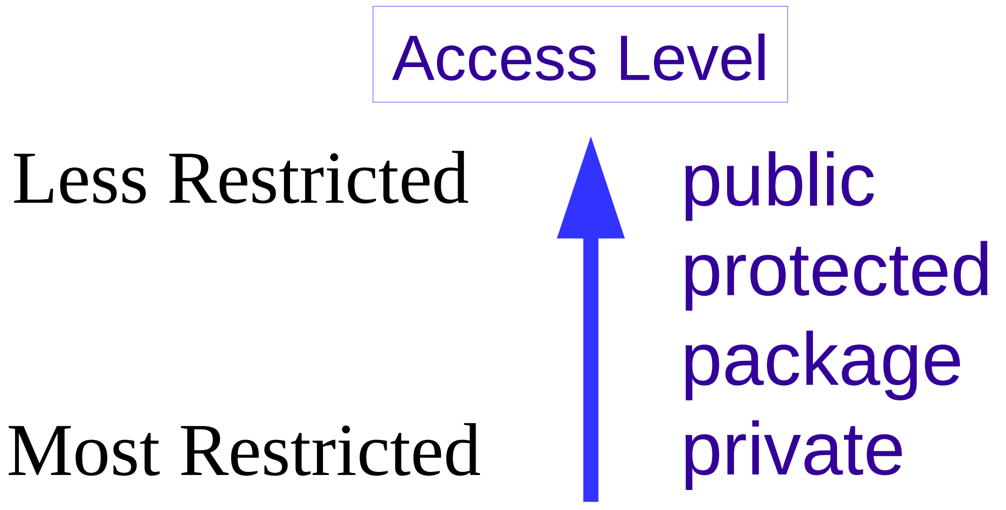
Object Superclass

All classes defined without an explicit `extends` clause automatically extend `Object`



More About Access Level

- Methods cannot be restricted in sub class:
- If you are overriding a method. If it is public in superclass, your cannot change it to private.



- final class and final methods

ObjectSuperclass

Most useful methods:

- String toString()
- boolean equals(Object otherObject)

Overriding the `toString` Method

- Returns a string representation of the object
- Useful for debugging:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
// Sets s to "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

Overriding the `tostring` Method

`toString` is called whenever you concatenate a string with an object:

```
"box=" + box;
// Result: "box=java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

`Object.toString` prints class name and the *hash code* of the object

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to something like "BankAccount@d24606bf"
```

Overriding the `toString` Method

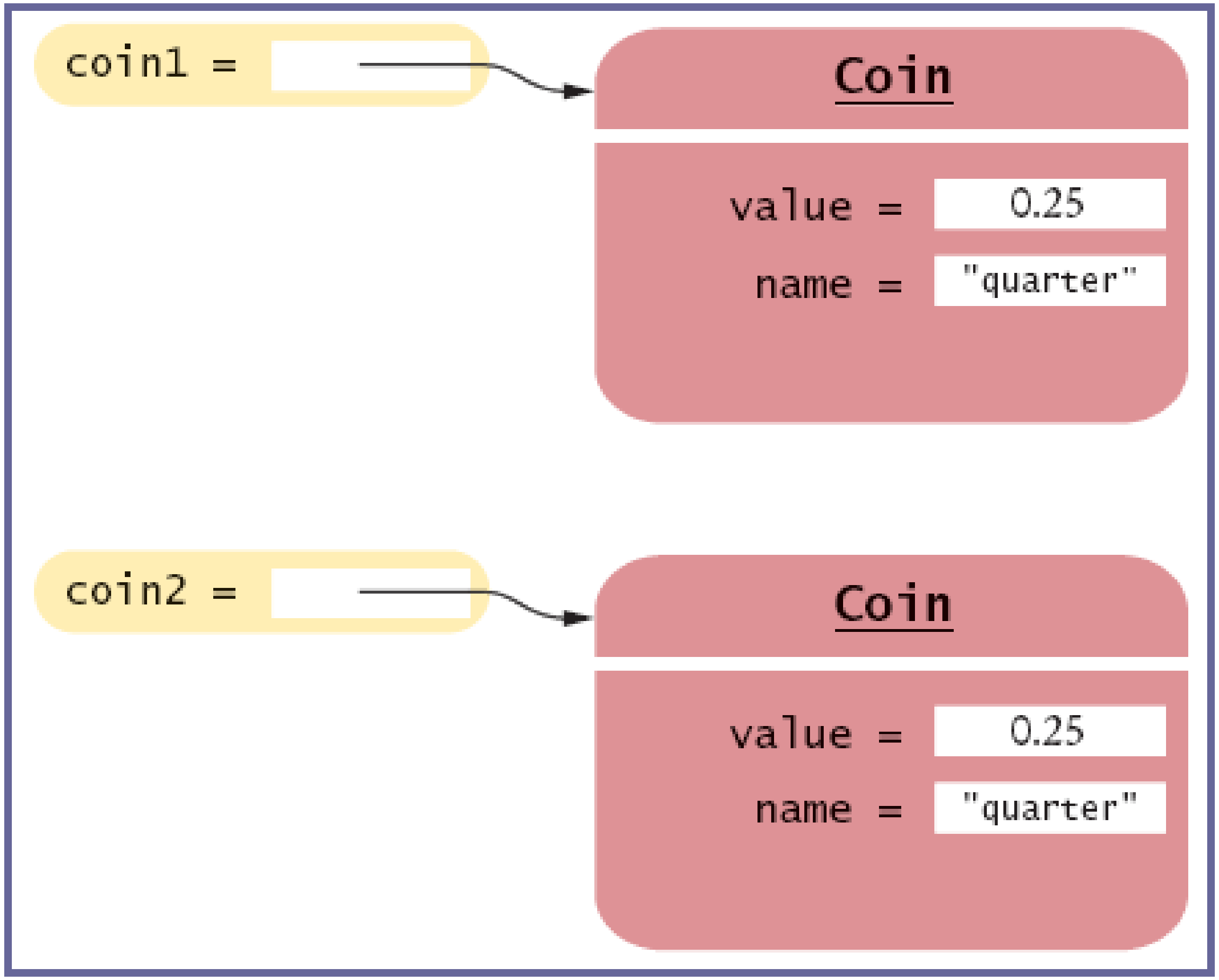
➢To provide a nicer representation of an object, override `toString`:

```
public String toString(){
   return "BankAccount[balance=" + balance + "]";
}
```

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to "BankAccount[balance=5000]"
```

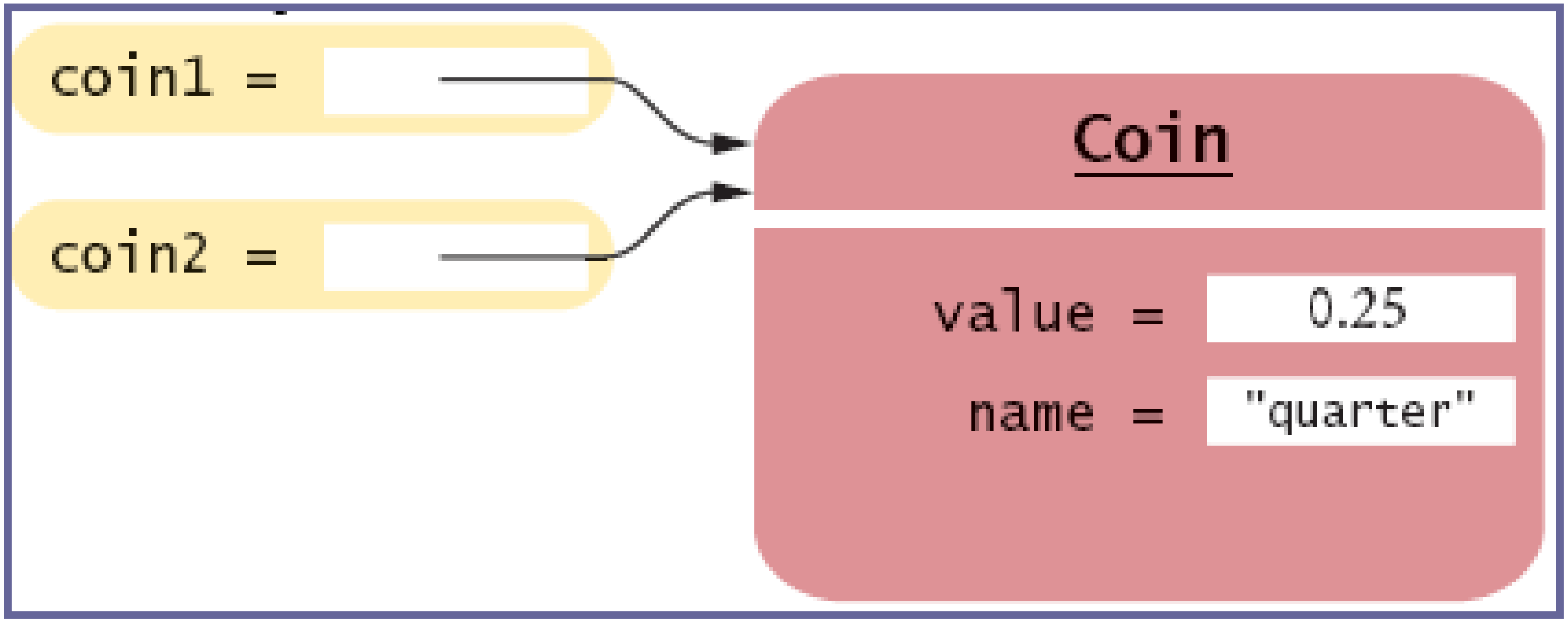
Overriding the `equals` Method

`equals` tests for equal *contents*



Overriding the `equals` Method

`==` tests for equal *location*



Overriding the `equals` Method

- Define the `equals` method to test whether two objects have equal state
- When redefining `equals` method, you cannot change object signature; use a *cast* instead:

```
public class Coin
{
   . . .
   public boolean equals(Object otherObject)
   {
      Coin other = (Coin) otherObject;
      return name.equals(other.name) && value == other.value;
   }
   . . .
}
```

Summary

- Inheritance is a is-a relation.
- An **abstract class** in Java is a class that is never instantiated. Its purpose is to be a parent to several related classes.
- An **abstract method** has no body. It just declares an *access modifier*, *return type*, and *method signature* followed by a *semicolon*.
- Abstract classes are a way of organizing a program.
- In Java, type of a variable doesn't completely determine type of object to which it refers (**Polymorphism** means "many forms.“)

Summery

Java has four levels of controlling access to fields, methods, and classes:

- public
- private
- protected
- package

- Most useful methods of class Object:
- String toString()
- boolean equals(Object otherObject)

  Good idea to override these methods in your classes