OOJ Lecture 10
Inheritance and Polymorphism

- Reading: Savitch, Chapter 7
- Reference: Big Java, Horstman, Chapter 9

Objectives

- To understand the common superclass Object
- To override Object's methods
  - toString()
  - equals()
  - clone()

Object: The Cosmic Superclass

- Class Object is the root of the class hierarchy.
- Every class has Object as a superclass.
- All classes are direct or indirect subclasses of Object

Object: The Cosmic Superclass

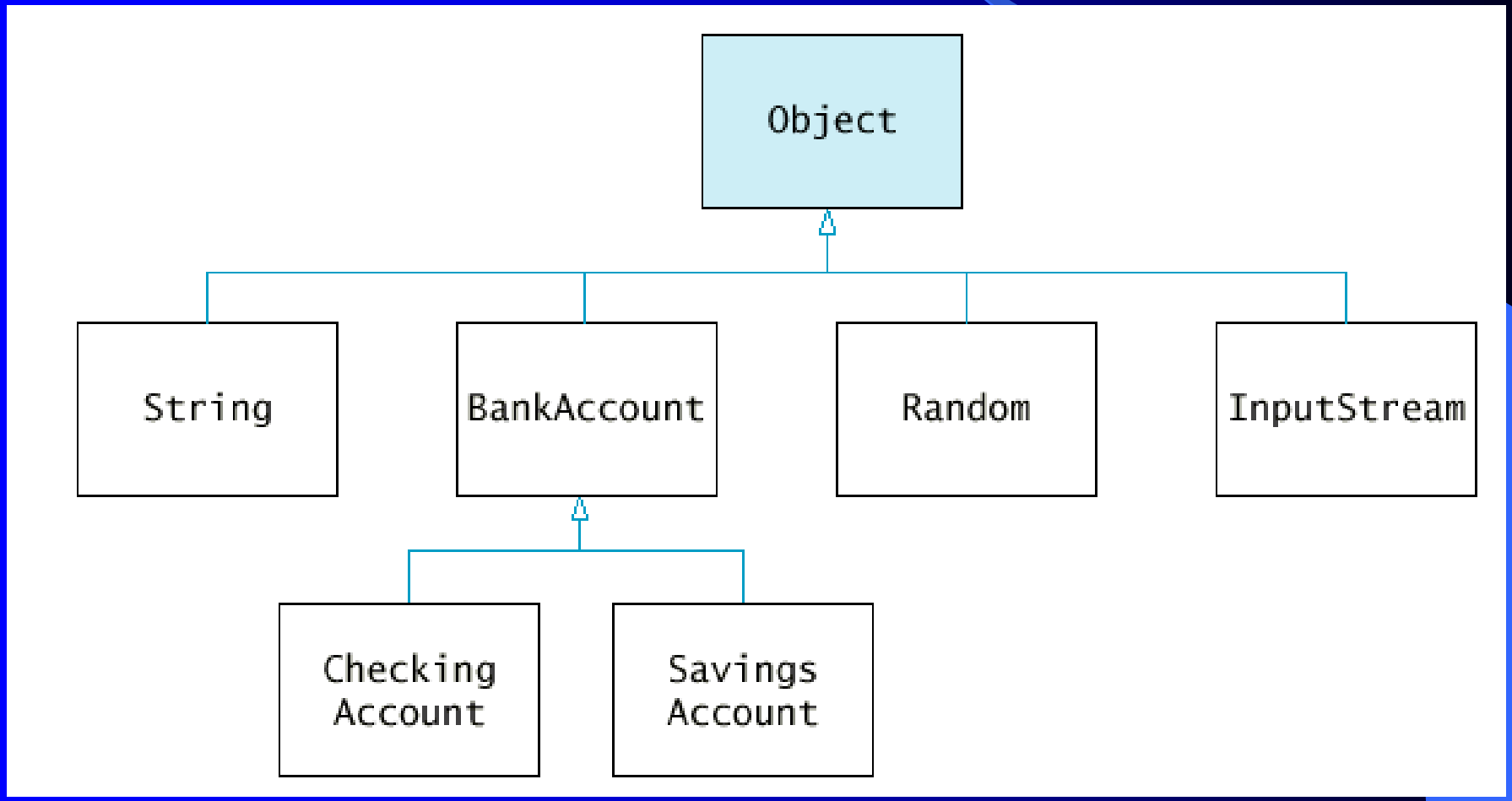
```
class java.lang.Object
{   public Object()
    public final Class getClass()
       //Returns the runtime class of an object.
    public boolean equals(Object obj)   //x.equals(y)
    public final void notify()
    public final void notifyAll()
       //Wakes up the thread(s) that is waiting on this
       //object's monitor.
    public final void wait(long timeout) throws
    InterruptedException
       //the current thread wait until the notify() method or a
       //specified amount of time has elapsed.

   protected Object clone() throws CloneNotSupportedException
   public String toString()     …
}
```

Object: The Cosmic Superclass

- Most useful methods:
  - `String toString()`
    - Returns a string representation of the object
  - `boolean equals(Object otherObject)`
    - tests for equal *contents* with another object
  - `Object clone()`
    - Make a full copy of an object
    - Copying object reference gives two references to same object

The Object Class is the Superclass of Every Java Class



The `toString` Method

- Returns a string representation of the object
- Useful for debugging
- Example (Horstmann Chapter 9):

```
Rectangle cerealBox = new
                    Rectangle(5,10,20,130);
String s = cerealBox.toString();
```

- Sets s to the value:

```
java.awt.Rectangle[x=5,y=10,width=20,height=30]
```

- `Rectangle` class has overridden method `toString` of `Object` to return a `String` representing this `Rectangle` and its values

The `toString` Method

- It is recommended that all subclasses override this method
- Example: If BankAccount doesn't override the `toString` method:

```
BankAccount momSaving = new
                    BankAccount(5000);
System.out.print(momSaving.toString());
```

- It prints class name and object address BankAccount@d2460bf, which is useless

The `toString` Method

- If toString is used by concatenation operator

```
aString + anObject
```

- means:

```
aString + anObject.toString()
```

- For Example:

```
System.out.print("cerealBox=" +cerealBox);
```

- Returns:

```
cerealBox = java.awt.Rectangle[x=5,y=10,
                               width=20,height=30]
```

Overriding the `toString` Method

- Override `toString`:

```
//Reference: Horstman, Chapter 9
public class BankAccount
{
    public String toString()
    {
      return "BankAccount[balance=" + balance + "]";
    }
       . . .
}
BankAccount momSaving = new BankAccount(5000);
System.out.print(momSaving);
```

- Will print: BankAccount[balance= 5000]

Supply `toString` to all classes

- It is helpful for checking errors.

- If you want to know information about an object and there is a toString method
  - you can insert a few print statements
    `System.out.print(aObject);`
  - and thus peek inside the object whilst the program is running.

Inheritance and `getClass` Method

- When inheriting by subclass, it's unnecessary to hard code the class name into any information
- Use the Object method `public final` **`Class`** `getClass()` to get a Class object
- Use the `public Sring getName()` method to get the name of the class

```
//Reference: Horstman, Chapter 9
public class BankAccount
{   public String toString()
    { return getClass().getName()+"[balance=" +
      balance + "]";
    }
    // The method will return the correct class
    // name when used by a subclass
}
```

Inheritance and `getClass` Method

- If a subclass SavingAccount uses the toString method

```
SavingAccount momSaving = new
                              SavingAccount(5000);
System.out.print(momSaving);
```

- It will print: `SavingAccount[balance = 5000]`
- Subclass SavingAccount can override the toString method of BankAccount, to print more information

```
//Reference: Horstman, Chapter 9
public class SavingAccount extends BankAccount
{  public String toString()
   { return super.toString() +"[InterestRate=" +
                                 interestRate + "]";
   }
}
```

Overriding the `equals` Method

```
public boolean equals(Object obj) {.. }
```

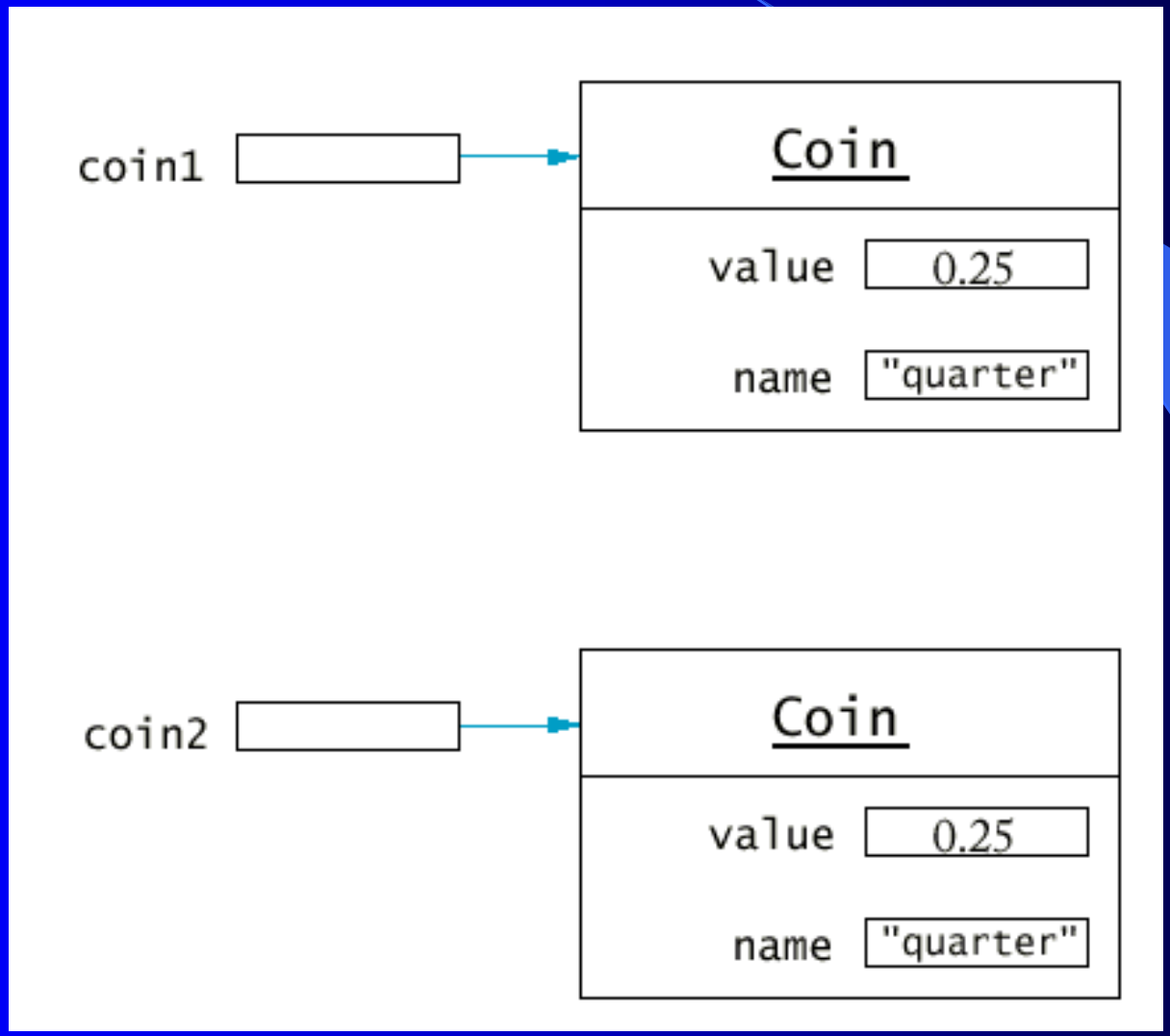
- What is the difference between method equals() and == comparison ?
- `equals` is to test two objects having same *contents*

```
if (coin1.equals(coin2)) …
```

- == is to test whether two references are to the same *location*

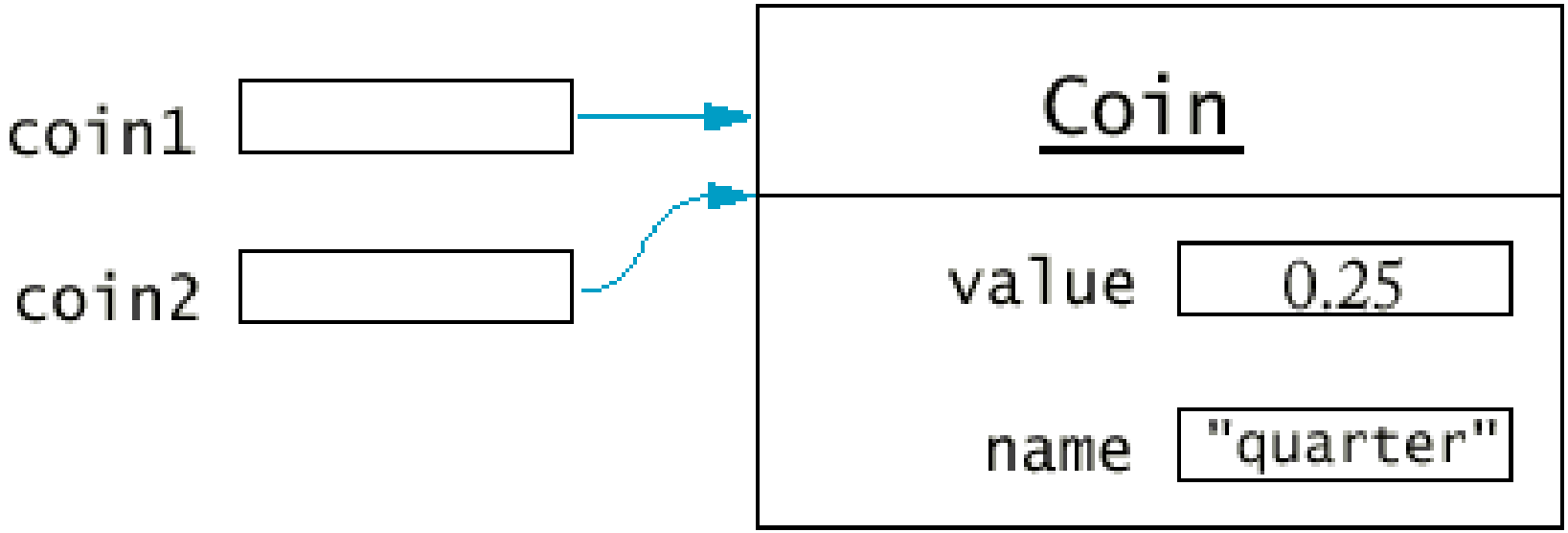
```
if (coin1 == coin2)
```

Two References to Equal Objects



Equals

Two References to Same Object



= =

Overriding the `equals` Method

- The Object class doesn't know of any subclasses
  - Must cast the `Object` parameter to subclass

```
//Reference: Horstman, Chapter 9
public class Coin
{   public boolean equals(Object otherObject)
    {
       Coin other = (Coin)otherObject;
       return name.equals(other.name) &&
                          value == other.value;
    }
}
```

- Use `equals(.)` to compare object fields
- Use == to compare two number fields

Using the equals Method

- You should test for the correct type before doing the comparison, otherwise:

```
Coin1.equals(x)
```

will cause the program to die if x is not a Coin object

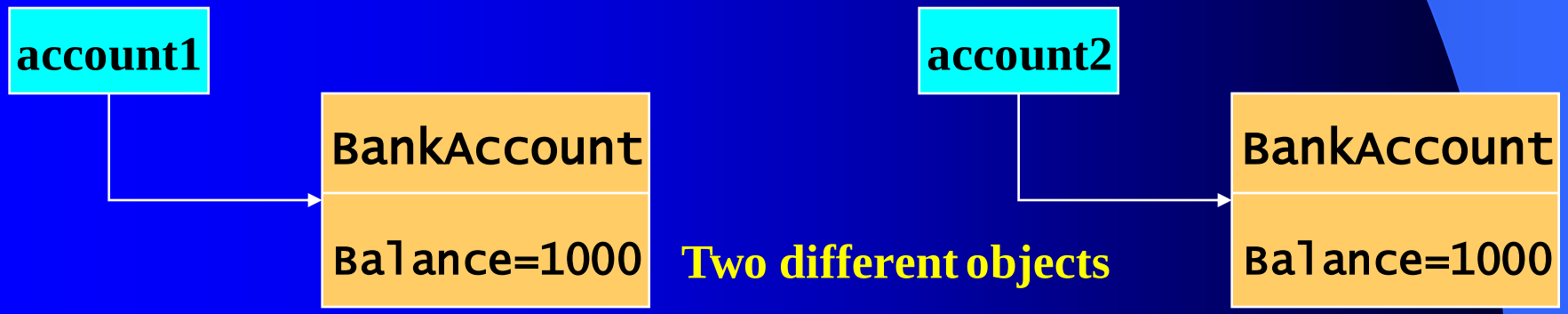
```
public boolean equals(Object otherObject)
{  if ((otherObject == null) || (getClass() !=
        otherObject.getClass()))
   return false;  // test whether they are the
   same type
   Coin other = (Coin)otherObject;
        return name.equals(other.name) && value
   == other.value;
}
```

Overriding the clone Method

- Clone creates and returns a copy of `this` object.
- Example: Using clone to make a copy of an object

```
BankAccount account1 = new BankAccount(1000)
BankAccount account2 = (BankAccount)account1.clone();
```

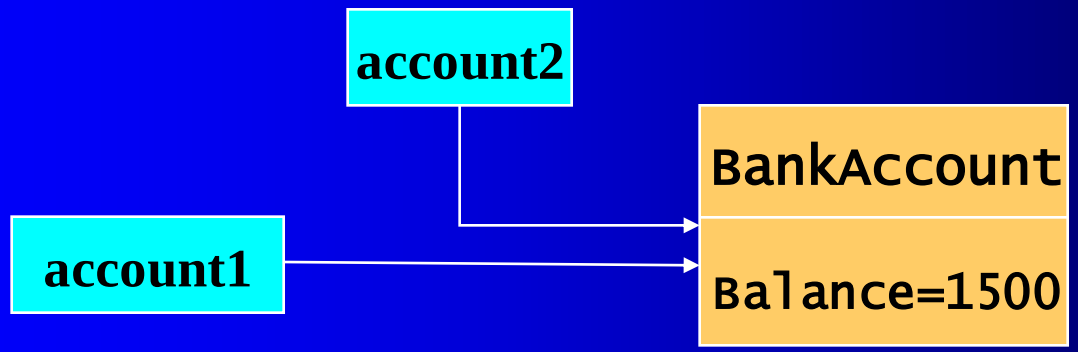
- Must cast to the BankAccount because return type is `Object` in clone method
- Protected Object clone() throws a `CloneNotSupportedException`



Overriding the clone Method

- Clone is different to copying two references
- Copying an object reference gives two references to the same object

```
BankAccount account1 = new BankAccount(1000)
BankAccount account2 = account1;
account2.deposit(500);
//both have $1500
```



Overriding the `clone` Method

- Define a `clone` method to make a new object:

```
public class BankAccount
{  public Object clone()
   {
      BankAccount cloned = new BankAccount();
      cloned.balance = balance;
      return cloned;
   }..
}
```

- Note: using this method requires a cast to BankAccount

```
BankAccount afund =
   (BankAccount)account1.clone();
```

Inheritance & the `clone` Method

- The `clone` method has limitations: it doesn't work for subclasses

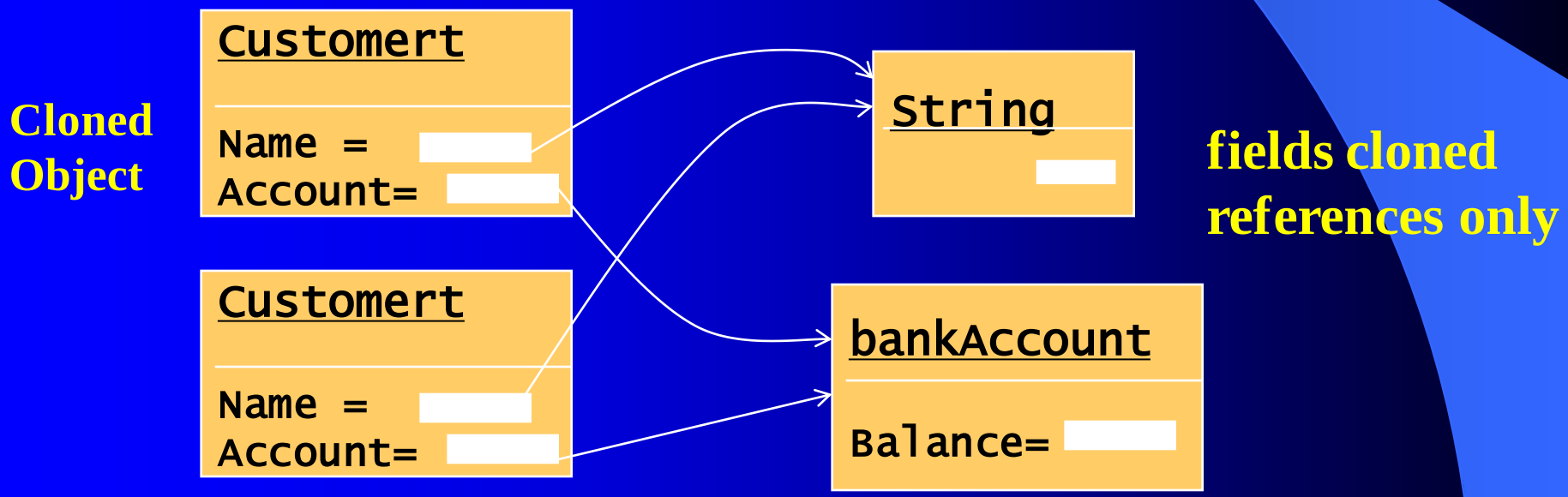
```
SavingAccount momSavings = new SavingAccount(10);
Object clonedAccount = momSavings.clone();
```

- In this case the clone method constructs a bank account not a saving account
- It is better to use the `Object.clone()` method for cloning

```
public Object clone()
{..
   Object clonedAccount = super.clone();
   return clonedAccount;
}
```

Inheritance & the clone Method

- The Object's clone method only clones one level:
  - If the cloned object has some object fields, only the object references are cloned, this is called a shallow copy



Inheritance & the clone Method

You must implement interface cloneable in order to use the clone methods of Object. Cloneable doesn't have a method.

```
public class Customer implements Cloneable
{   public Object clone( )
    {  try
       {      //call Object.clone, account is a reference only
              Customer cloned = (Customer)super.clone( );
              cloned.account= (BankAccoutn)account.clone( );
              //account is copied
              return cloned;
              //cloned customer with cloned account
       }
       catch(CloneNotSupportedException e)
       {      //This should not happen
              return null;      //To keep the compiler happy.
       } private String name; private  BankAccount account;  }
}
```

Interface Cloneable

```
public interface Cloneable{}
```

- Implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.

- Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.

- Classes that implement this interface should override Object.clone (which is protected) with a public method.

Inheritance & the `clone` Method

To use the clone method correctly:

1. The class must implement the `cloneable` interface
2. It must override the `clone` method of `Object`
3. The clone method must use a try-catch block to catch exceptions - `CloneNotSupportedException`