Java Coding – OOP Part 2

To object or not…

Object References

- An object variable is a variable whose type is a class
  - Does not actually hold an object.
  - Holds the memory location of an object

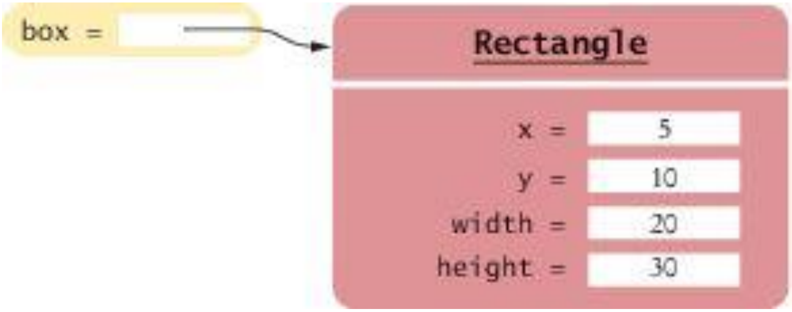


Figure 15 An Object Variable Containing an Object Reference

Object References

- **Object reference:** describes the location of an object
- After this statement:

  ```
  Rectangle box = new Rectangle(5, 10, 20, 30);
  ```

  - Variable `box` refers to the `Rectangle` object returned by the `new` operator
  - The `box` variable does not contain the object. It refers to the object.

Object References

- Multiple object variables can refer to the same object:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box;
```

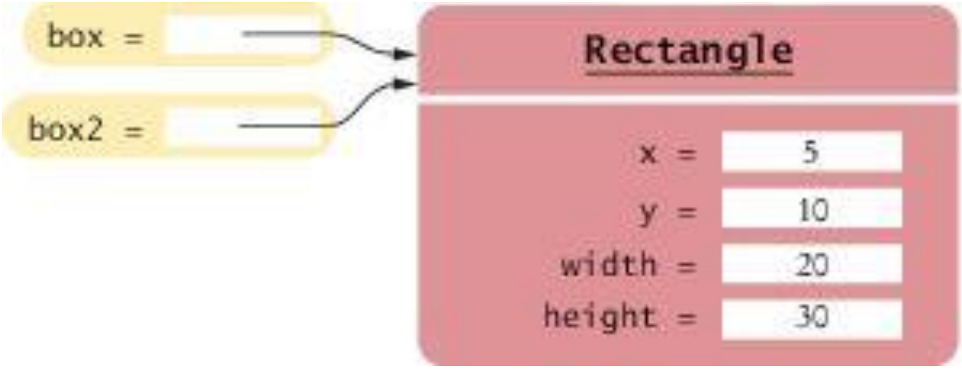


Figure 16 Two Object Variables Referring to the Same Object

Copying Object References

- When you copy an object reference
  - both the original and the copy are references to the same object

```
Rectangle box = new Rectangle(5, 10, 20, 30);❶
Rectangle box2 = box;❷
box2.translate(15, 25);❸
```

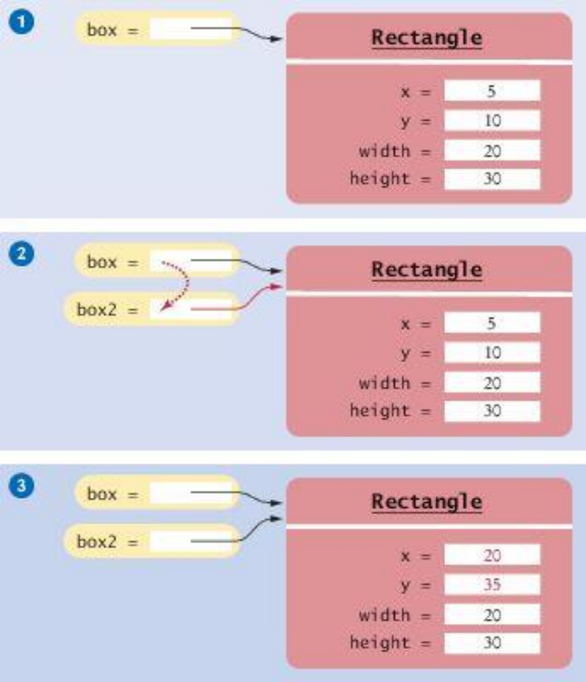
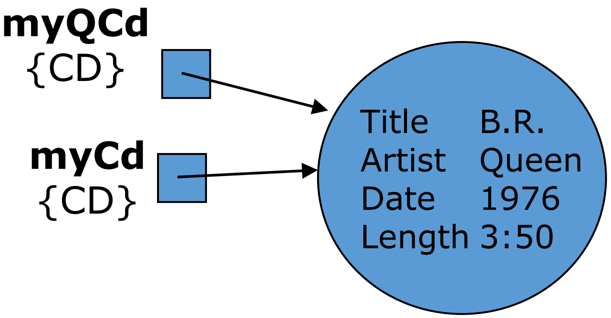


Figure 19 Copying Object References

Same or Different? (1)

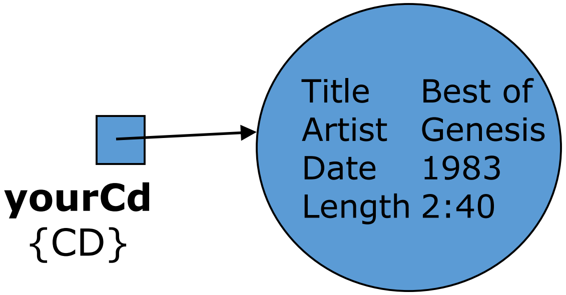
- Comparing objects



myQCd
{CD}

myCd
{CD}

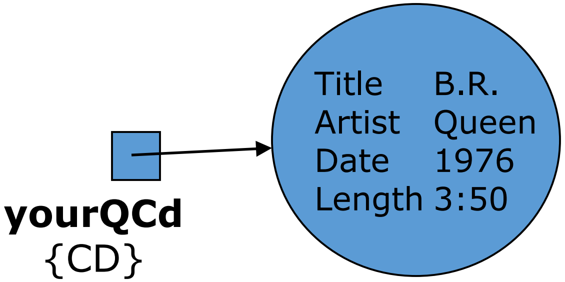
Title B.R.
Artist Queen
Date 1976
Length 3:50



yourCd
{CD}

Title Best of
Artist Genesis
Date 1983
Length 2:40

```
if ( myCd == yourCd)
   System.out.println( "Same");
else
   System.out.println( "Different");
```

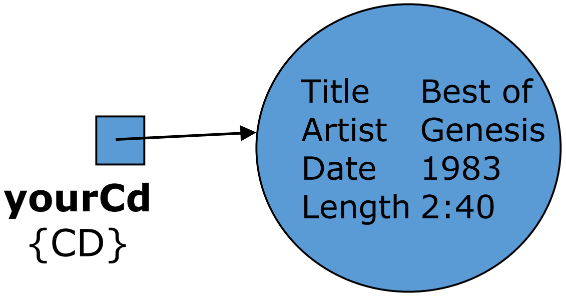


yourQCd
{CD}

Title B.R.
Artist Queen
Date 1976
Length 3:50

```
if ( myCd == yourQCd)
   System.out.println( "Same");
else
   System.out.println( "Different");
```

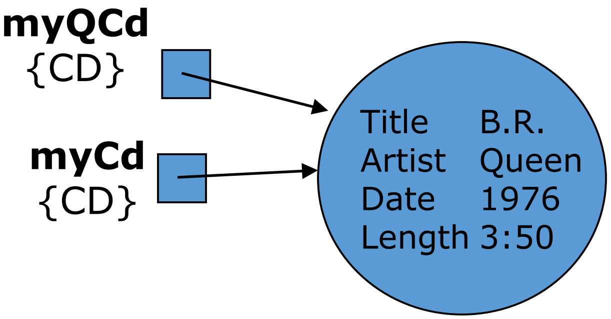
Same or Different? (1)



yourCd
{CD}

Title Best of
Artist Genesis
Date 1983
Length 2:40

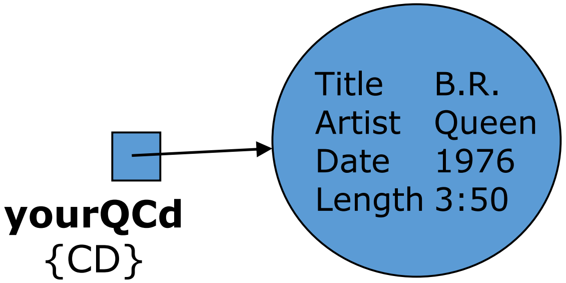
- Comparing objects



myQCd
{CD}

myCd
{CD}

Title B.R.
Artist Queen
Date 1976
Length 3:50



yourQCd
{CD}

Title B.R.
Artist Queen
Date 1976
Length 3:50

- “==“ is comparing references, not the object properties
- “==“ says whether the references refer to the same individual object or to two distinct objects

- Only “myCd == myQCd” would give true

Same or Different? (2)

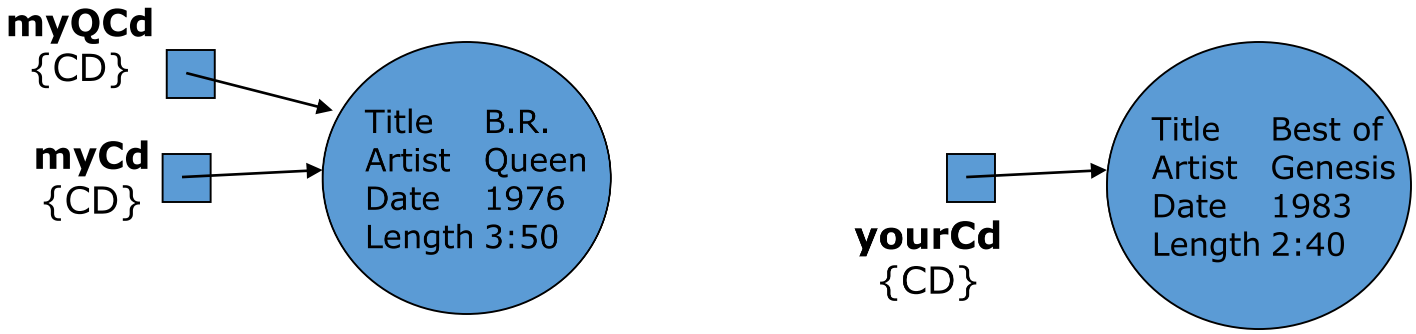
Define an “equals” method to compare objects

- Can write an equals method in CD class that compares CD’s by content, not reference
  - “myCd.equals(myCd)” would give true!

- Write such a method
- You could name the method anything you want
  - “equals” is the **convention** Java uses… so follow it!

Same or Different? (2)

- Define an "equals" method



myQCd
{CD}

myCd
{CD}

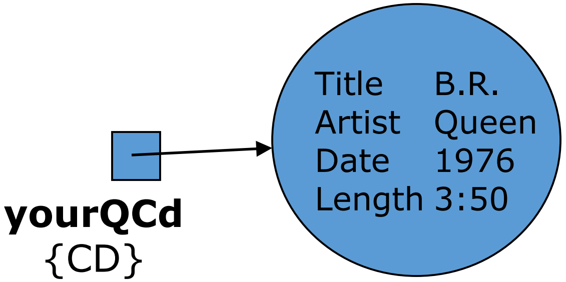
Title B.R.
Artist Queen
Date 1976
Length 3:50

yourCd
{CD}

Title Best of
Artist Genesis
Date 1983
Length 2:40

```
if ( myCd.equals( yourCd) )
   System.out.println( "Same");
else
   System.out.println( "Different");

if ( myCd.equals( yourQCd) )
   System.out.println( "Same");
else
   System.out.println( "Different");
```



Title B.R.
Artist Queen
Date 1976
Length 3:50

yourQCd
{CD}

Copying

- copying has different semantics for primitive and object type data

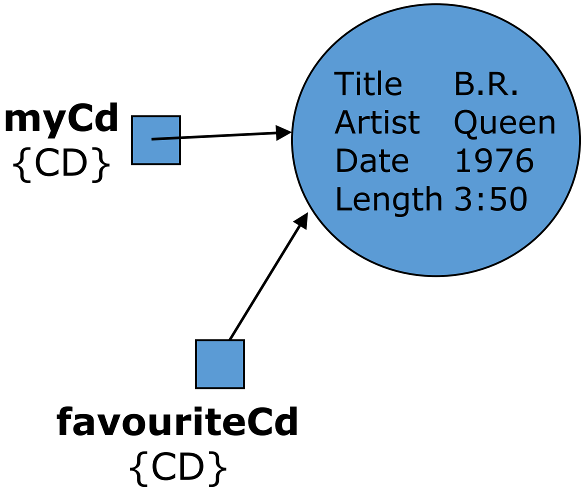
```
int i, j;
i = 5;
j = i;
i++;
Sys… ( i, j);
```

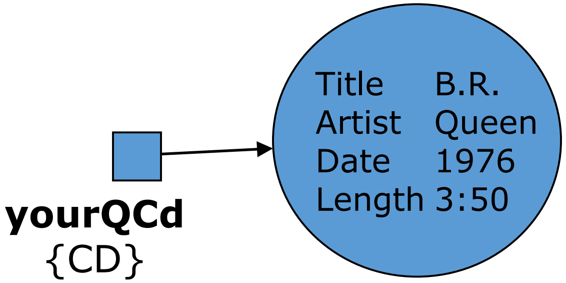
Different

```
Person me, x;
me = new Person( …);
x = me;
me.setComments( “nice!”);
Sys… ( me.getComments()
          + x.getComments(), );
```

Same!

Copy vs. Clone





```
favouriteCd = myCd;
```

```
yourQCd = myCd.clone();
```

```
// inside the CD class write a clone method

public CD clone(){
return new CD(title,artist,date,length);}
```

Copy vs. Clone

- Copying only copies the reference, making the copy refer to the same object
- Clone involves creating an entirely new object and copying all the properties of the first into it
- Java automatically provides a clone method for all objects
  - BUT be careful, it performs a “shallow” copy which is fine for primitive types
  - Not necessarily for embedded objects (which end up shared by both the original and clone objects!)
  - Doing clone() properly is a problem since it requires implementing clonable & handling exceptions!)
- **Use copy constructors as an alternative**
  - **For example, yourQCd = new CD(myCd);**

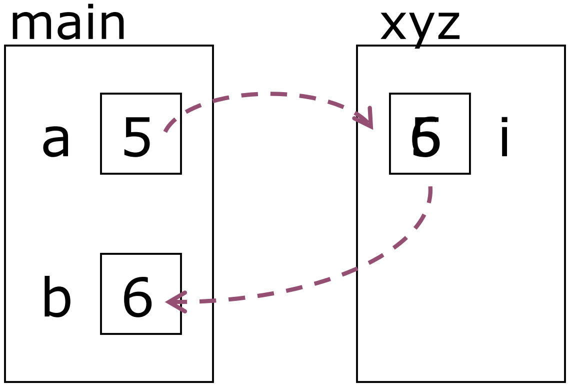
Parameter Passing (1)

- Primitive types…

```
public int xyz( int i) {
        i++;
        return i;
}
```

main

```
int a, b;
a = 5;
b = xyz( a);
Sys… ( a, b);
```



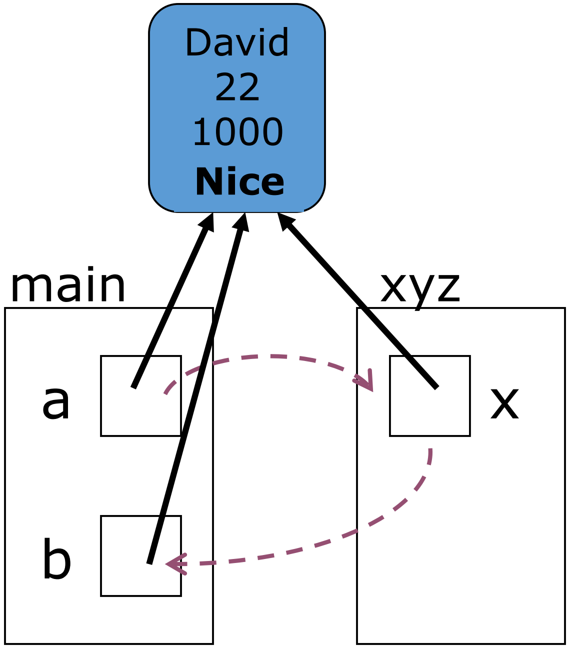
Parameter Passing (2)

- Object types…

```
public Person xyz( Person x) {
    x.setComments(“Nice”);
    return x;
}
```

main

```
Person a, b;
a = new Person( “David” …);
b = xyz( a);
Sys… ( a.getComments()
          + b.getComments() );
```



NOTICE – changing the properties of the object referred to by the formal parameter in the method DOES change the properties of the corresponding (actual parameter’s) object in the main method

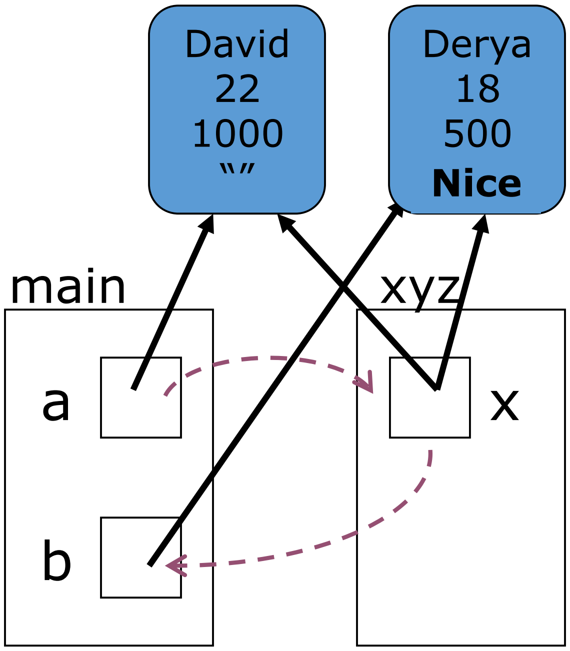
Parameter Passing (3)

- Object types…

```
public Person xyz( Person x) {
    x = new Person( “Derya” …);
    x.setComments(“Nice);
    return x;
}
```

main

```
Person a, b;
a = new Person( “David” …);
b = xyz( a);
Sys… ( a.getComments()
          + b.getComments() );
```



NOTICE – changing the reference of the formal parameter in the method DOES NOT change the corresponding actual parameter’s reference in the main method.

All Objects…

- automatically have
  - boolean equals(Object)
  - Object clone()
  - String toString()

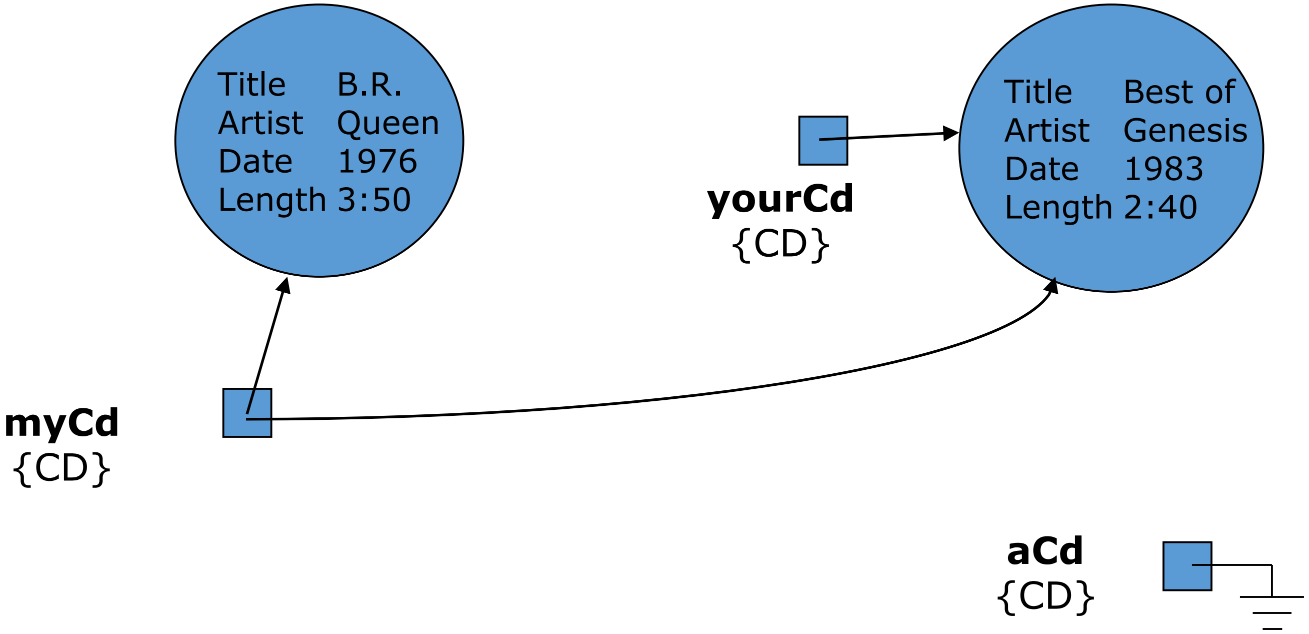
Code using these methods will compile & run even if your class does not define them!

- BUT
  - they may not do what you would like/expect, so implement yourself!

equals() defaults to “==“
clone() defaults to “shallow copy”
toString() defaults to “classname@hashvalue”

Lost objects & null

- Java collects its garbage!



Title B.R.
Artist Queen
Date 1976
Length 3:50

Title Best of
Artist Genesis
Date 1983
Length 2:40

yourCd
{CD}

myCd
{CD}

aCd
{CD}

```
myCd = yourCd;
```

```
aCd = null;
```

Lost objects & null

- What happens when “myCd = yourCd;” is executed?
- Variable only refers to one object at a time.
- So my Queen CD is lost
- Objects having no references to them cannot be used!
- They are effectively garbage
- Java automatically collects such garbage allowing the space to be reused/recycled for other objects

Lost objects & null

- **“null” is a special value** that can only be applied to references
- Can compare references to null
    - e.g. if (aCd == null) or if (myCd != null)
- Cannot compare references using <, >, <=, >=
    - (or add, subtract or multiply them!)

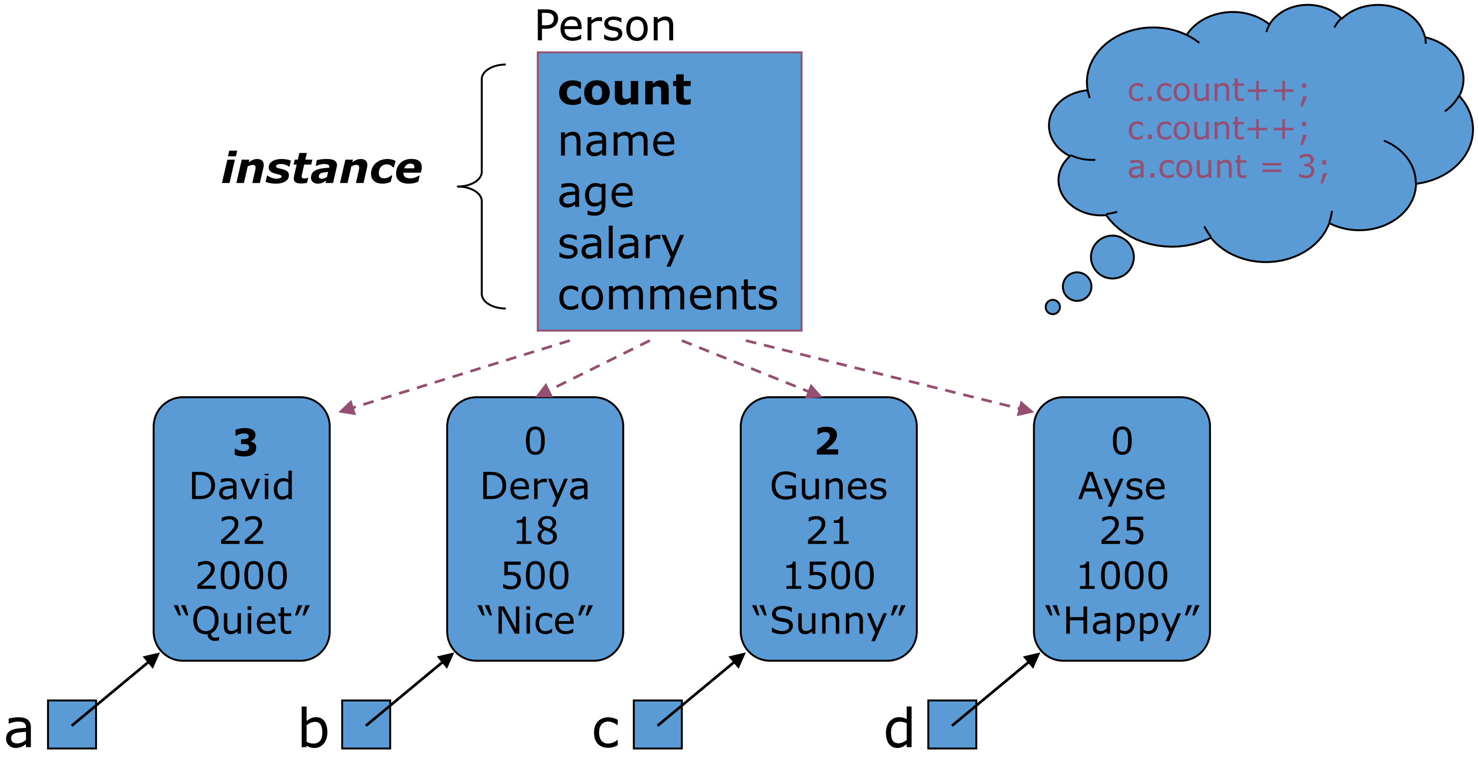
- Attempting to access the properties or methods of an object that doesn’t exist because the reference is null, results in a “**nullPointerException**”

Static vs. Instance

Static vs. instance Variables

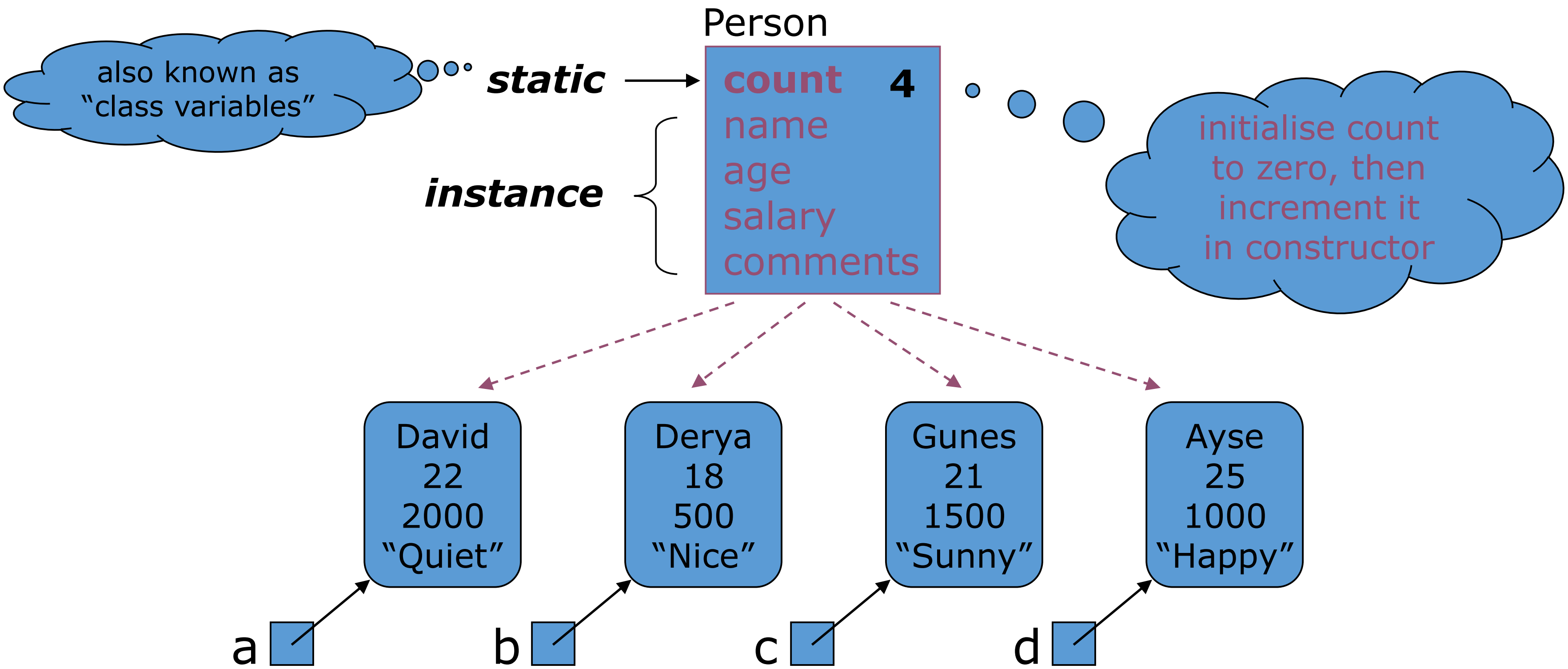
- "count" as an instance variable

  -- each instance (object) has count variable



Static vs. instance Variables

- "count" as a static variable
  -- only one count variable, associated with class



Static vs. instance Variables

- Each instance of the class, i.e. each individual object, has its own values for each instance variable
- But there is only ever one copy of a static variable (also called a class variable)

- **Static data accessible via classname.variablename (and object.variablename)**
- **Instance data only accessible via object.variablename syntax**

- Static data can be accessed even if there are no instances of the class.
- Same goes for static methods, e.g. a getCount() method here or the main method!
- **Static methods can only refer to static data and data defined locally in method. Why?**

- Static data used
  - for constant definitions (outside method but in class) –never changes so only need one copy!
  - for singletons (classes which allow one and only one object to be created.)

Misc:

- Can combine, so static “nextID” gives next value to be assigned to instance variable “personID”

- Constants often defined as static

  hence saving space

```
public static final int PI = 3.142;
public static final String COMPANY = “Bilkent University”;
```

Static vs. instance Methods

- Classes can have both
  static & instance methods.
- Static methods useful when
  - **accessing static variables**

    public static int getCount()
  - **object state is not needed**

    public static int getAge(day, month, year)
- Static methods

  cannot access instance variables or methods
- Instance methods

  can access static & instance, variables & methods

Singletons *(design pattern)*

- **Problem**: Ensure only a single instance of a class is created.
  (for database or network connections, etc.)
- **Solution**: Combine static variable, private constructor & static method!

```
public class SingletonClass {

      private static SingletonClass ourInstance = new SingletonClass();

      private SingletonClass() {
      }

      public static SingletonClass getInstance() {
            return singletonObj;
      }
}
```

Singletons *(design pattern)*

- Implemented by creating a class with a method that creates a new instance of the class if one does not exist

- If an instance already exists, it simply returns a reference to that object

- To make sure that the object cannot be instantiated any other way, the constructor is made private or protected

Singletons *(design pattern)*

```
public class ClassicSingleton {
   private static ClassicSingleton instance = null;
   protected ClassicSingleton() {
      // Exists only to defeat instantiation.
   }
   public static ClassicSingleton getInstance() {
      if(instance == null) {
         instance = new ClassicSingleton();
      }
      return instance;
   }
}
```

The `this` Reference

- Two types of inputs are passed when a method is called:
  - The object on which you invoke the method
  - The method arguments
- In the call `momsSavings.deposit(500)` the method needs to know:
  - The account object (`momsSavings`)
  - The amount being deposited (`500`)
- The **implicit parameter** of a method is the object on which the method is invoked.
- All other parameter variables are called **explicit parameters**.

The `this` Reference

- Look at this method:

  ```
  public void deposit(double amount)
  {
     balance = balance + amount;
  }
  ```

  - `amount` is the explicit parameter
  - The implicit parameter(`momSavings`) is not seen
  - `balance` means `momSavings.balance`
- When you refer to an instance variable inside a method, it means the instance variable of the implicit parameter.

The `this` Reference

- The `this` reference denotes the implicit parameter

  ```
  balance = balance + amount;
  ```

  actually means

  ```
  this.balance = this.balance + amount;
  ```

- When you refer to an instance variable in a method, the compiler automatically applies it to the `this` reference.

The this Reference

- Some programmers feel that inserting the this reference before every instance variable reference makes the code clearer:

```
public BankAccount(double initialBalance)
{
   this.balance = initialBalance;
}
```

The `this` Reference

- The `this` reference can be used to distinguish between instance variables and local or parameter variables:

```
public BankAccount(double balance)
{
   this.balance = balance;
}
```

- A local variable shadows an instance variable with the same name.
  - You can access the instance variable name through the `this` reference.

The `this` Reference

- A method call without an implicit parameter is applied to the same object.
- Example:

```
public class BankAccount
{
   . . .
   public void monthlyFee()
   {
      withdraw(10); // Withdraw $10 from this account
   }
}
```

- The implicit parameter of the `withdraw` method is the (invisible) implicit parameter of the `monthlyFee` method

The this Reference

- You can use the this reference to make the method easier to read:

```
public class BankAccount
{
   . . .
   public void monthlyFee()
   {
      this.withdraw(10); // Withdraw $10 from this account
   }
}
```