





Xi'an Jiaotong-Liverpool University

西交利物浦大學

CPT111 Java Programming

Lecture 7

Introduction to Objects

Previously, in the last few weeks:

- Arrays and Loops
 - To store useful **information** in arrays
 - To store **values** in arrays - **primitives and objects**
 - To use **loops to navigate** through arrays
- Methods
 - Break code up into blocks
 - Using public static methods
 - Create smaller main methods

This Week:

- Objects
 - Representing the world as objects
- Constructor
 - Populating the instance variables
- Public and private
 - Should variables and methods be shared or kept private?
- Encapsulation
 - Data hiding
- Using objects within a array with for loops
- Static vs Non-static
- Comparing two objects

Object Oriented Programming

Representing things as objects

Primitives don't have methods

- Primitive data types (**i**nt, **d**ouble, **b**oolean etc) store an actual value
 - The variable holds that value
- Primitives have no methods
 - Only the value

Wrapper Classes

```
Integer.toString(3;)
```

- Week 5, we introduced "Wrapper Classes"
- Useful for converting between String, Integers etc.
- A way of treating primitives as objects. We wrap the primitive to access useful methods
- Although an int is a primitive, an Integer allows us to treat it as an object, and so there are a number of methods, such as Compare(), toString() etc. that we can access.

Objects have methods

- Objects (**S**tring, **S**canner, **M**ath) are referred to by reference
 - The variable holds a **memory address** pointing to the object
- The object contains **properties** (variables) and **behaviour** (methods)

Objects

- **Java** is an **object oriented** language
 - Nearly everything in Java is an object.
- **The world** is an **object oriented** world
 - Nearly everything in the world is an object.
- Real-world objects have **states** and **behaviour**

Objects

- Objects in Java often represent objects in the real world (or eg a game)
- These objects have **state**: eg a character in a game could have
 - Location
 - Points
 - Strength
 - Weapons
 - *States can change*
- They would also have **behaviour**:
 - Walk
 - Fight
 - This ***behaviour*** changes the ***state***

Objects – key points

- Can store everything we need in objects
 - Objects have Attributes - variables
 - Objects have Methods
 - Anything can be an object!
-
- Better to group connected data together as an object

Previously used Java Objects

String and Scanner

Existing Java Objects

- We have encountered several different objects so far. This includes String, Math, and Scanner objects
- The String object holds a piece of data – text – and has **methods** to **process** it. What sort of things
- Similarly, the Scanner and Math objects also have data and methods

String Object

```
public class UsingStrings {  
    public static void main(String[] args) {  
  
        String someText;  
        someText = new String();  
        someText = "Hello Andrew";  
        System.out.println(someText);  
    }  
}
```

- The variable **someText** is declared and holds an object of the **String** class.
- **new String()** creates – **instantiates** - a new **String** object
- This object is given a value of: "Hello Andrew"

String Methods

- `myString.length();`
- `myString.toLowerCase();`
- `myString.toUpperCase();`
- `myString.equals(anotherString);`
- `myString.equalsIgnoreCase(anotherString);`
- `myString.contains("java");`

Scanner Object - Review

- The Scanner object is a useful way to receive the `System.in` character stream, i.e. accessing characters that you type
 - Like a String, a Scanner is an object
 - It contains a number of useful methods
- Its purpose is to receive and handle keyboard input

Objects Summary

- Objects can have methods
- Can store data
- Many pre-written objects
 - Code written by someone
 - Stored as a java library
 - (Can import as needed)
 - Have methods you can access
- You can make your own objects!

Data Objects

Creating your own objects

Data or Real-World Objects

- A very common type of object in Java is a **data object**
- Often represents a real-world object
 - such as a Person
 - It does not represent a whole Person
 - It holds the pieces of data we are interested in.
- We can write a java class to represent this object

Classes in Java

- Currently we have been working in a single class
 - Everything is **static**, a static class
 - This means that a class is just a class
 - When you run it, theres one version of it
- However, we can use the same class to create multiple objects

The Person *class*

Defines *Objects* to hold simple data
about a person

Person Data Object

- If we want to store information about people, we can create Person objects
- Each Person has a number of different properties
 - Name
 - Sex
 - City
 - Date of birth
 - Id number
- We can store these in Java

A Person class

```
package personproject;
```

```
public class Person {  
    String name;  
    String gender;  
    String city;  
    String dateOfBirth;  
    int idNumber;  
}
```

So, how do we create all these different Person objects with their different data?

- The class defines the data that each Person object will hold
- There will be many Person objects (one for each person!)
- Each Person object will contain different data
- Each Person *object* is an **instance** of the Person *class*
- These variables are called **instance variables**

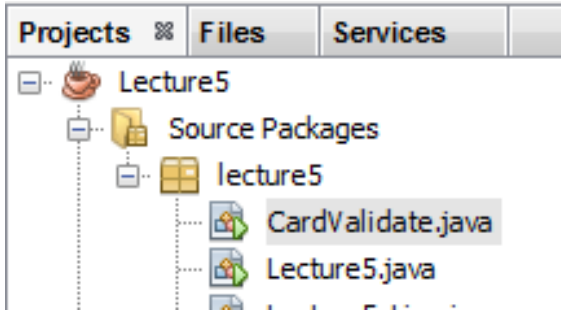
A Person class

```
public class Person {
```

- We start by declaring
- public class _____,
and open with {
- Similar to the class
files you have been
creating so far
- Best to create it in a
new Java file

A Person class

```
package personproject;
```



```
~/
package lecture5;

/**
 *
 * @author ANDREW.ABEL
 */
public class CardValidate {
```

- You might see this
- The package refers to a group of java classes.
- It matches the name of the grouping
- It is useful for much larger projects with multiple files
- If you use the “default package”, then no package will be listed
- Generally, you do not need to worry about this!

Instance Variables

```
String name;  
String gender;  
String city;  
String dateOfBirth;  
int idNumber;
```

- Remember, each person will be different, so they will store different values
- Note, these are not static!
- If they are static then no matter how many objects you create, they will overwrite
- Each Person object will contain different data.
- Each Person *object* is an **instance** of the Person *class*.
- These variables are called **instance variables**.

Our class so far

```
public class Person {  
    String name;  
    String gender;  
    String city;  
    String dateOfBirth;  
    int idNumber;  
}
```

- No methods
- No main methods
- How do we give each object values?

Constructors

Making distinct objects

Constructor

- An object has special methods called “Constructors”
- Can be used to assign values when an object is created (“constructed”)
- Can also carry out any initial tasks
- It can have parameters
 - Whatever parameters you want!
 - Can be used to assign values to the instance variables

Creating a constructor

- In our Person class, we created a number of class variables
- But we did not give them values
- As they are objects, each one we create can have different variable values
- It is the constructor that allows our initial objects to be different when they are first created!
- This is how we create different people with the same class

```
//constructor  
public Person(String name, String gender,  
              String city, String DoB, int id) {
```

Constructor

- When an object is created, it looks for a constructor
- A class can have more than one constructor
- Should have at least one
- It will run this method whenever a new object is created
- This is how we can pass variables into the object

A Constructor has no return type

The Constructor's name is exactly the same as the class name. So it is the only method to start with an uppercase letter.

```
//constructor  
public Person(String name, String gender,  
String city, String DoB, int id) {
```

The Constructor takes one argument for each of the object's data fields. (It doesn't have to, but in our case it should.)


```
public class Person {
```

```
    String name;  
    String gender;  
    String city;  
    String dateOfBirth;  
    int idNumber;
```

```
//constructor
```

```
public Person(String name, String gender,  
               String city, String DoB, int id) {  
    this.name = name;  
    this.gender = gender;  
    this.city = city;  
    this.dateOfBirth = DoB;  
    this.idNumber = id;  
}
```

- The Constructor has an argument for each object of data; for each instance variable.
- These are **local variables**, remember?
- So we assign the values to the actual **instance variables**, like this:

The Person class has no main method. That will be in a separate controller class:

```
public class Person {
```

```
    String name;
```

```
    String gender;
```

```
    String city;
```

```
    String dateOfBirth;
```

```
    int idNumber;
```

```
//constructor
```

```
public Person(String name, String gender,  
               String city, String DoB, int id) {
```

```
    this.name = name;
```

```
    this.gender = gender;
```

```
    this.city = city;
```

```
    this.dateOfBirth = DoB;
```

```
    this.idNumber = id;
```

```
}
```

- The Constructor has an argument for each piece of data; for each instance variable.
- These are **local variables**, remember?
- So we assign the values to the actual **instance variables**, like this:

```
public class Person {
```

```
    String name;
```

```
    String gender;
```

```
    String city;
```

```
    String dateOfBirth;
```

```
    int idNumber;
```

```
    //constructor
```

```
    public Person(String name, String gender,  
                  String city, String DoB, int id) {
```

```
        this.name = name;
```

```
        this.gender = gender;
```

```
        this.city = city;
```

```
        this.dateOfBirth = DoB;
```

```
        this.idNumber = id;
```

```
    }
```

- The Constructor has an argument for each piece of data; for each instance variable.
- These are **local variables**, remember?
- So we assign the values to the actual **instance variables**, like this:

```
public class Person {
```

```
    String name;
```

```
    String gender;
```

```
    String city;
```

```
    String dateOfBirth;
```

```
    int idNumber;
```

```
    //constructor
```

```
    public Person(String name, String gender,  
                  String city, String DoB, int id) {
```

```
        this.name = name;
```

```
        this.gender = gender;
```

```
        this.city = city;
```

```
        this.dateOfBirth = DoB;
```

```
        this.idNumber = id;
```

```
    }
```

- The Constructor has an argument for each piece of data; for each instance variable.
- These are **local variables**, remember?
- So we assign the values to the actual **instance variables**, like this:

Our class so far

```
public class Person {  
    String name;  
    String gender;  
    String city;  
    String dateOfBirth;  
    int idNumber;  
  
    // Constructor  
    public Person(String name, String gender,  
        String city, String dOB, int id){  
        this.name = name;  
        this.gender = gender;  
        this.city = city;  
        this.dateOfBirth = dOB;  
        this.idNumber = id;  
    }  
}
```

- Declare a class
- No main method
- Declare instance variables
- Create a constructor

this.objectInstanceVariable

Referring to an object variable

This Reference

- In our constructor, we have 2 different variables with the same name
- One was created as **an instance variable** and can be seen anywhere in the class
- One is created within the constructor, and contains the argument passed in, and **only exists within the constructor**
- This is confusing
- How can we solve this?

```
public Person(String name) {  
    this.name = name;  
}
```

This Reference

- One option
- Use different variable names!

```
public Person(String inputName) {  
    name = inputName;  
}
```


This Reference

- Other option, use `this.name`
- When we use `this.`, we refer to the instance variable
- So we can access this object's instance/class variables
- `this.name` is the class variable
- `name` is the local variable;

```
public Person(String name) {  
    this.name = name;  
}
```

This Reference

- Using **this._____** refers to this particular object
- It can refer to its own variables and methods
- Can call any of its **own methods** within the class
- Or any of its **own variables**
 - e.g. `int a = this.name;`
- Very useful in constructors!

```
this.name = "A call to the object";  
this.getName();
```

Our class so far

```
public class Person {  
    String name;  
    String gender;  
    String city;  
    String dateOfBirth;  
    int idNumber;  
  
    // Constructor  
    public Person(String name, String gender,  
        String city, String dOB, int id){  
        this.name = name;  
        this.gender = gender;  
        this.city = city;  
        this.dateOfBirth = dOB;  
        this.idNumber = id;  
    }  
}
```

- Declare a class
- No main method
- Declare instance variables
- Create a constructor

Creating a Person Object

Creating from our Main class

Constructing a Person object

```
public class PersonProject {
```

main() is NOT in the Person class.

```
    public static void main(String[] args) {
```

```
        Person p1;
```

```
        p1 = new Person("James Bond", "M",  
                        "London", "1921-11-11", 007);
```

```
        Person p2 = new Person("Jackie Chan", "M",  
                                "Hong Kong", "1954-4-7", 1);
```

In the Person class:

```
//constructor
```

```
public Person(String name, String gender,  
              String city, String DoB, int id) {
```

Creating our people

- We use our Person class to create objects
- In our main method, we create a new person
- Just like when we created a new Scanner etc.
- We choose arguments that match the constructor

```
public static void main(String[] args) {  
  
    Person p1 = new Person("James Bond", "M",  
        "London", "1921-11-11", 007);  
  
}
```

We now have two different classes

Our **main** class has the main() method. This is sometimes called the **controller** class. It is (usually) static. Only one main class exists in the application. It has no **constructor**.

The **main** class can have or contain lots of objects, data objects such as Person objects. Each Person object holds data for one person.

Our Person class is not static. It defines Person data objects: their data fields and methods. It has a constructor used to construct Person data objects, each with **different** data. Many different Person data objects will exist.

So now our Person class defines a very basic data object (actually right now it is more of a structure, or struct: it has no methods!

Creating our people

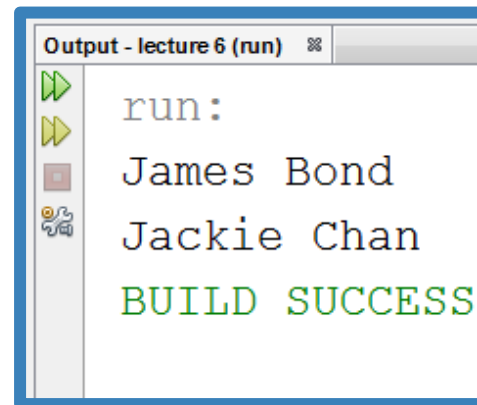
- Can use the same class to now create multiple people
- The same class can create different objects

```
public static void main(String[] args) {  
  
    Person p1 = new Person("James Bond", "M",  
        "London", "1921-11-11", 007);  
  
    Person p2 = new Person("Jackie Chan", "M",  
        "Hong Kong", "1954-4-7", 1);  
  
}
```


Creating our people

- Can access our data using the data fields in person
- Note that the Person code is unchanged
- But by storing things as an object, we can have multiple objects using the same class

```
System.out.println(p1.name);  
System.out.println(p2.name);
```

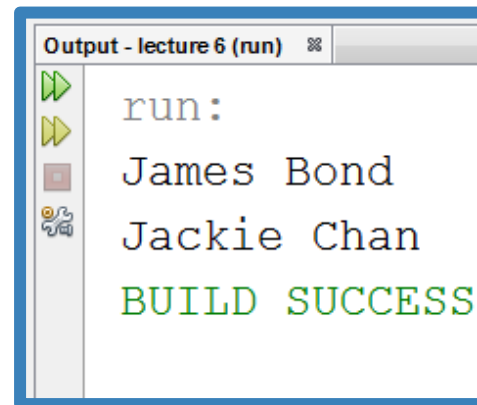


Creating our people

- Can access our data using the data fields in person
- Note that the Person code is unchanged
- But by storing things as an object, we can have multiple objects using the same class

```
System.out.println(p1.name);  
System.out.println(p2.name);
```

Right now, anyone can access the data fields directly. However, anyone can also change them. We do not want that!



Public and Private

Data hiding

Public and Private

- So far, we haven't really cared
- Public or private, both work within a class
 - can create **public void**, or **private void** methods
- With multiple classes, there IS a difference
- If it is **PRIVATE**, can only be accessed by that class
 - Useful for internal methods
 - something that only needs to be done **within the objects**
- If it is **PUBLIC**, then it can be accessed by other classes
 - Useful for object methods that need to be accessed

Public and Private

- As a good habit, things should be private by default for data security
- variables, methods, should be private, unless we need to access it from a separate class
- We need to access our constructor and any other methods that we need
 - They should therefore be public !

Encapsulation

- all attributes should be private
- Internal workings should be private
- Object should only share what it needs
- Can use getter methods to access

```
private String name;
```

```
public String getName() {  
    return name;  
}
```

```
public class Person {  
    private String name;  
    private String gender;  
    private String city;  
    private String dateOfBirth;  
    private int idNumber;  
}
```

Make the fields private

Private instance variables and private methods can only be accessed from within the same class. OK, great: but now no-one can read the data in our data object!

```
public String getName() {  
    return name;  
}  
  
public String getGender() {  
    return gender;  
}  
  
public String getCity() {  
    return city;  
}  
  
public String getDateOfBirth() {  
    return dateOfBirth;  
}  
  
public int getIdNumber() {  
    return idNumber;  
}
```

- Add accessor methods, or 'getters'
- Public methods to return private data.
- Hiding the data and other internal workings of the class and objects, and only accessing them through public methods, is called encapsulation.
- Encapsulation is one of the core principles of OOP, and Java.

We use the getter methods to get the data:

```
public class PersonProject {  
  
    public static void main(String[] args) {  
        Person p1;  
        p1 = new Person("James Bond", "M",  
                        "London", "1921-11-11", 007);  
        Person p2 = new Person("Jackie Chan", "M",  
                                "Hong Kong", "1954-4-7", 1);  
  
        System.out.println("Name = "+p1.getName());  
    }  
}
```

run:

Name = James Bond

So, back in our main class, we can access the data fields in our data objects using the 'getter' methods. But we can't change the data

We use the getter methods to get the data:

```
public static void main(String[] args) {  
  
    Person p1 = new Person("James Bond", "M",  
        "London", "1921-11-11", 007);  
  
    Person p2 = new Person("Jackie Chan", "M",  
        "Hong Kong", "1954-4-7", 1);  
  
    System.out.println(p1.getName());  
    System.out.println(p2.getName());  
}
```

We can use the “getter” methods to access our variables from the main class

A little history...

Previously: we would also have 'setter'; and these are still common. However, the trend today is to have **immutable** objects: objects that cannot be changed after they are created. If you want to change any data, you have to make a new object

```
public void setName(String name) {  
    this.name = name;  
}  
  
public void setGender(String gender) {  
    this.gender = gender;  
}  
  
public void setCity(String city) {  
    this.city = city;  
}
```

Why? Parallel or concurrent processing is much more common today, and immutable objects are safer for this

Can add **checking** for valid values **before** setting the value

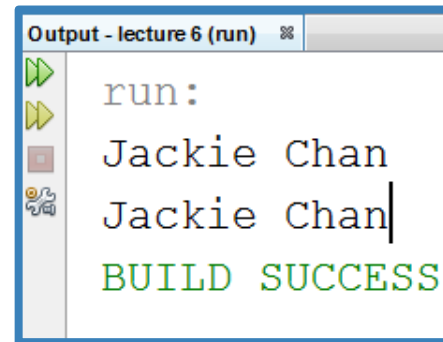
Static

- What about the **static** keyword?
- Static means that there is only one instance for the entire class
- So even if we create 100 objects, if we use a static variable for person name, then each person will have the same name
- If it is static, each object will overwrite the previous variable
- Can be useful

Static

```
public static void main(String[] args) {  
  
    Person p1 = new Person("James Bond", "M", "London",  
                           "1921-11-11", 007);  
  
    Person p2 = new Person("Jackie Chan", "M", "Hong Kong",  
                           "1954-4-7", 1);  
  
    System.out.println(p1.getName());  
    System.out.println(p2.getName());  
}
```

```
public class Person {  
  
    private static String name;  
    private String gender;  
    private String city;  
    private String dateOfBirth;  
    private int idNumber;  
}
```

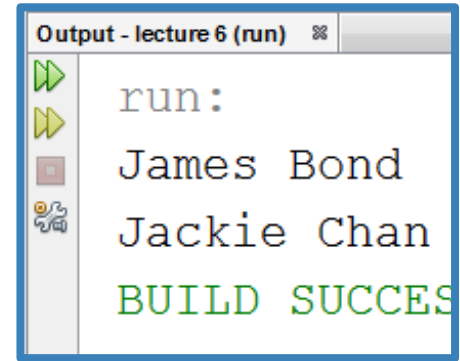


Static, so second
person declaration
overwrites first,
meaning BOTH are
Jackie Chan

Non-Static

```
public static void main(String[] args) {  
  
    Person p1 = new Person("James Bond", "M", "London",  
                           "1921-11-11", 007);  
  
    Person p2 = new Person("Jackie Chan", "M", "Hong Kong",  
                           "1954-4-7", 1);  
  
    System.out.println(p1.getName());  
    System.out.println(p2.getName());  
}
```

```
public class Person {  
  
    private String name;  
    private String gender;  
    private String city;  
    private String dateOfBirth;  
    private int idNumber;  
}
```



```
Output - lecture 6 (run) x  
run:  
James Bond  
Jackie Chan  
BUILD SUCCESS
```

Not static so both names are stored as separate instance variables, and are different

Use of Static

- Our **main method** is static, as there is only one main method in the entire program
- In previous weeks, our methods were **static**, as we were working with a single class, and not objects
- Our previous variables were static for the same reason
- But with our People, we are working with objects, so we do **not** use static

Use of Static – Good use

- Where might we use static?
 - If we want to count how many objects created
 - How many people objects?
- Can create in main method
- OR can use a static variable to count each...

What's next?

- We now have a functioning data class
 - Private fields (instance variables)
 - Public getters to access them
 - A constructor to create new data objects
- It can store all data about a person
 - A way of grouping information about 1 person together
 - Better than lots of variables, or even arrays!
- We will add more features, for a really good data object

Primitive variables and Object variables

`==` and `.equals()`

Variables pass a *value* or a *reference*

- int and double are variables holding **primitive** data
- A String is a variable pointing to an **object** (String)
- What is the difference?
- Primitive variables *hold a value*
- Object variables *hold a reference, a pointer to the memory address of the object*

```
int myNumber = 7;  
int newNumber = myNumber;  
myNumber ++;
```

Here, the **value** of myNumber (7) is assigned to newNumber. The two variables are completely unrelated and unconnected

Objects vs primitives

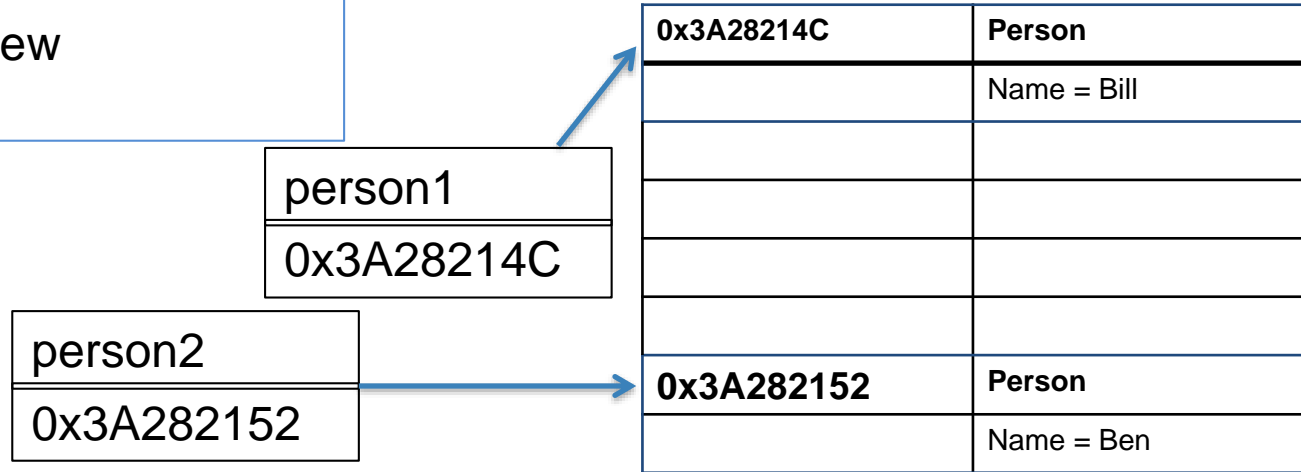
```
Person[] people = new Person[2];  
people[0] = p1;  
people[1] = p2;  
System.out.println("Array is " + people);
```

- An array is not a primitive
- when you try to print out whole array without looping through it
- Outputs the reference

Array is
[Lecture5.Person;@15db9742

Object variables: pointers

```
Person person1 = new  
Person("Bill");  
Person person2 = new  
Person("Ben");
```



Object variables hold the address in memory of the object. They are completely separate from the object, and do not know or hold any of the object's values. They 'point' to an address in memory; in some languages they are called pointers.

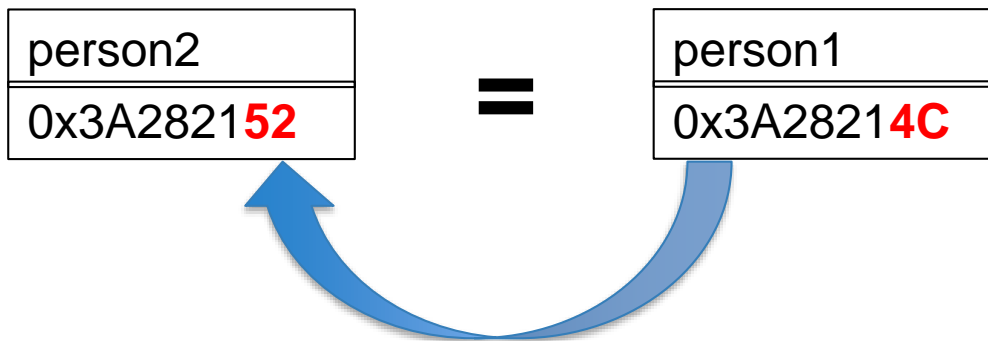
Pass by reference

person1
0x3A28214C

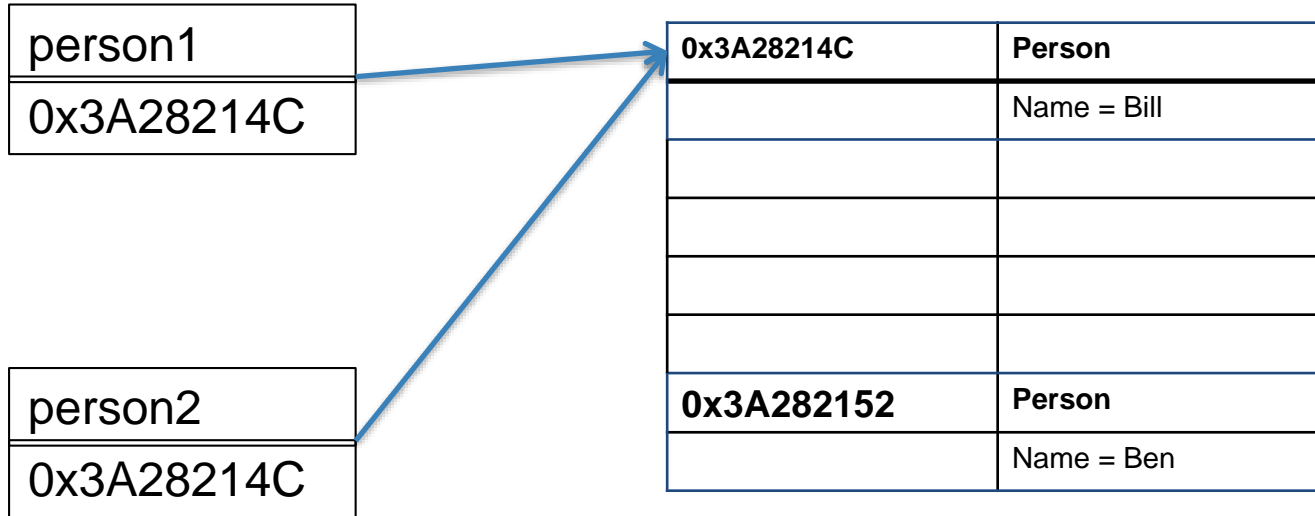
person2
0x3A282152

person2 = person1;

The memory address – **reference** – is copied from person1 to person2. **No values are copied at all.**



Now person1 and person2 point to the same Person object



== and .equals()

double number1		double number2	
3.214	=	3.214	True

Primitive variables hold a value. == compares the values, and returns TRUE if they are the same.

```
Person p1 = new  
Person("John");  
Person p2 = p1;
```

p1		p2	
0x00000001	=	0x00000001	True

Object variables hold an address. == compares the address, and returns TRUE if they are the same
(You could say that it is comparing the value of the address)

== and .equals()

```
Person p1 = new  
Person("John");  
Person p2 = p1;
```

p1
0x00000001

=

p2
0x00000001

True

- What if they hold the same values, but different addresses?
- i.e. all the values are the same, but are stored as different variables?

```
Person p1 = new Person("James Bond", "M", "London", "1921-11-11", 007);  
  
Person p2 = new Person("James Bond", "M", "London", "1921-11-11", 007);  
  
System.out.println(p1 == p2);
```

How do we compare our Person objects?

.equals()

- In Java, objects have an .equals() method.
 - E.g. Strings `variable.equals("variable")`
- In Java, we can use inheritance to “override” methods, and we can create an equals method
 - Inheritance will be discussed next week
 - .equals() is more advanced Java and will **not** be covered in this module (covered in ***CPT204 Advanced OOP***)

Same Person

- We can make an assumption that two people are the same if their name, id, and gender are the same
- We can therefore create a method
 - Return a boolean (true if same/false if different)
 - Use the getters to compare
 - Compare 'this' object

```
public boolean samePerson(Person that) {  
    return true;  
}
```

Same Person

- Two people, "this" and "that"
- We can compare “this” person i.e. the object, to “that” person, i.e. the person passed in
- Use the getters
- `Person p1 = new Person("James Bond", "M", "London", "1921-11-11", 007);`
- `Person p2 = new Person("Jackie Chan", "M", "Hong Kong", "1954-4-7", 1);`
- `boolean sameP = p1.samePerson(p2);`

```
if(!this.name.equals(that.getName())) {  
    return false;  
}
```

Same Person

- `Person p1 = new Person("James Bond", "M", "London", "1921-11-11", 007);`
- `Person p2 = new Person("Jackie Chan", "M", "Hong Kong", "1954-4-7", 1);`
- `boolean sameP = p1.samePerson(p2);`

Same Person

- `this.name = "James Bond"`
`that.getName() = "Jackie Chan"`
- If they are not equal, return false

```
if(!this.name.equals(that.getName())){  
    return false;  
}
```

Same Person

- If any attributes are not matching return false
- If all attributes match, return true

```
public boolean samePerson(Person that){  
    if(!this.name.equals(that.getName())){  
        return false;  
    }  
    if(!this.gender.equals(that.getGender())){  
        return false;  
    }  
    if(this.idNumber != that.getIdNumber()){  
        return false;  
    }  
    return true;  
}
```


The `.toString()` method

Displaying information

Anything else?

- Remember that objects are pointers, not values
- All java objects also have a toString() method.
- By default, it is not very useful.

```
System.out.println("Person 1 " + p1 );
```

```
Person 1  
lecture5.Person@15db9742
```

Anything else?

- We need to define a toString method in our data class.
- It should print the values of all the data fields - instance variables – in the object

```
public String toString() {  
    return "Person: name=" + name  
        + ", gender=" + gender  
        + ", city=" + city  
        + ", idNumber=" + idNumber;  
}
```

```
public class PersonProject {  
    public static void main(String[] args) {  
        Person p1;  
        p1 = new Person("James Bond", "M",  
                        "London", "1921-11-11", 007);  
        Person p2 = new Person("Jackie Chan", "M",  
                                "Hong Kong", "1954-4-7", 1);  
        System.out.println("p1 = "+p1.toString());  
        System.out.println("p2 = "+p2);  
    }  
}
```

run:

p1 = Person: name=James Bond, gender=M, city=London, idNumber=7

p2 = Person: name=Jackie Chan, gender=M, city=Hong Kong, idNumber=1

Looking good!

- Now we have a Person data class. It has:
 - Private data fields – instance variables
 - Constructor
 - Public getter methods
 - It is encapsulated
 - No setter methods
 - It is immutable
 - toString() method to print the object's values

Combining with Arrays

- Create an array of 3 people:

```
Person[] people = new Person[3];
```

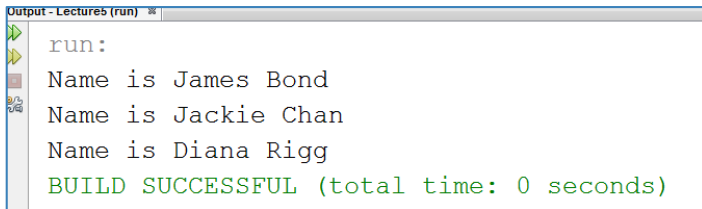
- Instantiate our array of people
- Can use our objects we created earlier, or create new ones

```
people[0] = p1;  
people[1] = p2;  
people[2] = new Person("Diana Rigg", "F", "London", "1943-8-21", 2);
```

For loops

- Now we can navigate through our array similar to before
- For example, if we want to show everyone's name

```
for (int i = 0; i < people.length; i++) {  
    System.out.println("Name is " + people[i].getName());  
}
```

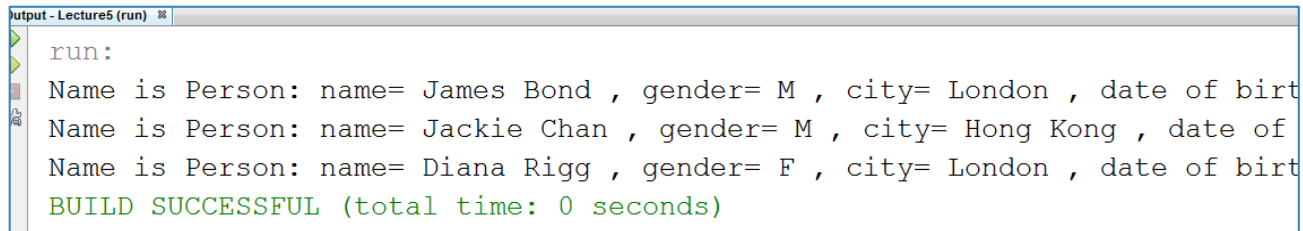


```
Output - Lectures (run)  
run:  
Name is James Bond  
Name is Jackie Chan  
Name is Diana Rigg  
BUILD SUCCESSFUL (total time: 0 seconds)
```

For loops

- Can use toString() method to display all

```
for (int i = 0; i < people.length; i++) {  
    System.out.println("Name is " + people[i].toString());  
}
```



```
run:  
Name is Person: name= James Bond , gender= M , city= London , date of birth= ...  
Name is Person: name= Jackie Chan , gender= M , city= Hong Kong , date of birth= ...  
Name is Person: name= Diana Rigg , gender= F , city= London , date of birth= ...  
BUILD SUCCESSFUL (total time: 0 seconds)
```


Thank you for your attention !

- In this lecture, you have learned about:
 - Objects, Attributes and Methods
 - Constructor and this.
 - Public vs Private
 - Static vs Non-static
 - Comparing two objects
 - Objects and Arrays
- Please continue to Lab 7 to complete Lab Tasks, and then solve
 - Exercise #7.1 - #7.4 and
 - CW1 #7.1 - #7.6