UNSW Business School
Information Systems and Technology Management

INFS1609/2609 Fundamentals of Business Programming

Lecture 9
Object-oriented Programming (2)

yenni.tim@unsw.edu.au

INFS1609/2609 Fundamentals of Business Programming

Topics for this week:

- Passing objects to methods
- Working with the `ArrayList` class
- Understanding the concept of inheritance
- To define a subclass from a superclass
- To invoke the superclass's constructors and methods using the `super` keyword
- To override instance methods in the subclass
- To distinguish differences between overriding and overloading
- Working with the `toString()` method

Main references
- *Textbook: Chapter 10 and 11*
- *Other online references posted on Ed*

INFS1609/2609 Fundamentals of Business Programming

A quick recap:

Creating objects, accessing data and using methods

[Supplementary Videos on Ed]

https://liveexample.pearsoncmg.com/html/TestSimpleCircle.html

Passing Objects to Methods

- Passing by value for primitive type value: the literal value is passed to the parameter
- Passing by value for reference type value: the value is the **reference** to the object

Passing objects to methods is passing the reference of the object

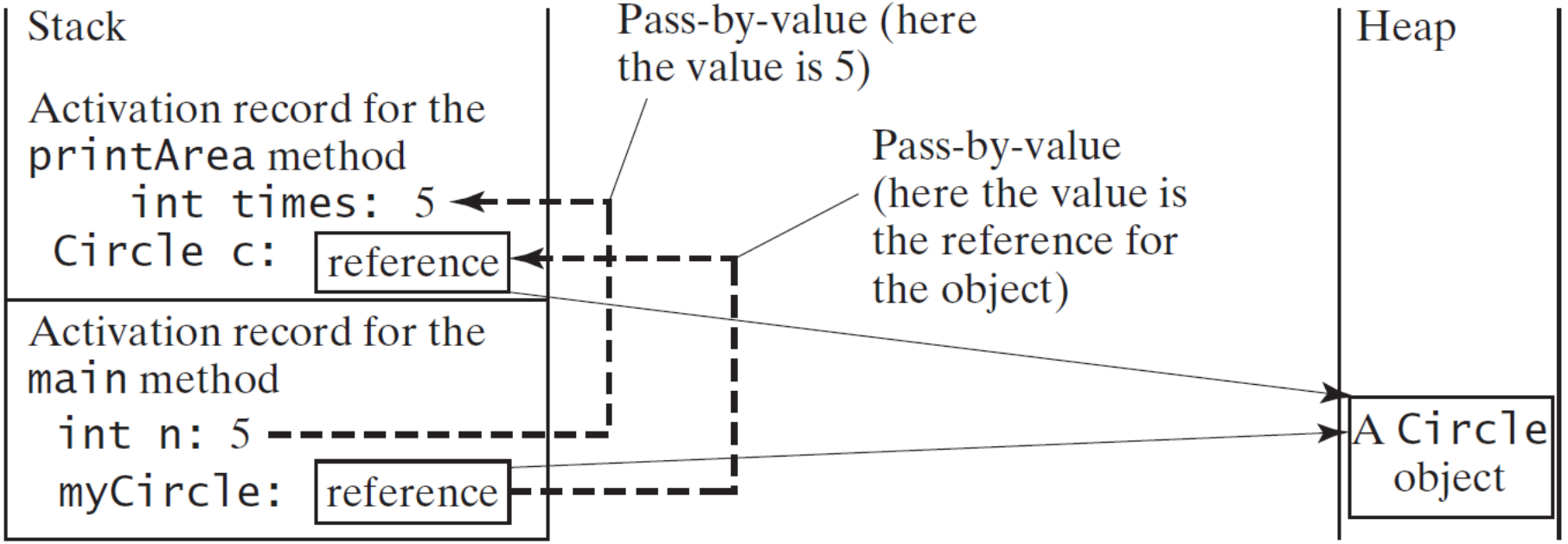
INFS1609/2609 Fundamentals of Business Programming

Passing Objects to Methods

https://liveexample.pearsoncmg.com/html/TestPassObject.html

Also read textbook page 372

Passing Objects to Methods



Passing Objects to Methods

What is the output of the following program?

```
public class Test {
  public static void main(String[] args) {
    Count myCount = new Count();
    int times = 0;

    for (int i = 0; i < 100; i++)
      increment(myCount, times);

    System.out.println("count is " + myCount.count);
    System.out.println("times is " + times);
  }

  public static void increment(Count c, int times) {
    c.count++;
    times++;
  }
}
```

```
class Count {
  public int count;

  public Count(int c) {
    count = c;
  }

  public Count() {
    count = 1;
  }
}
```

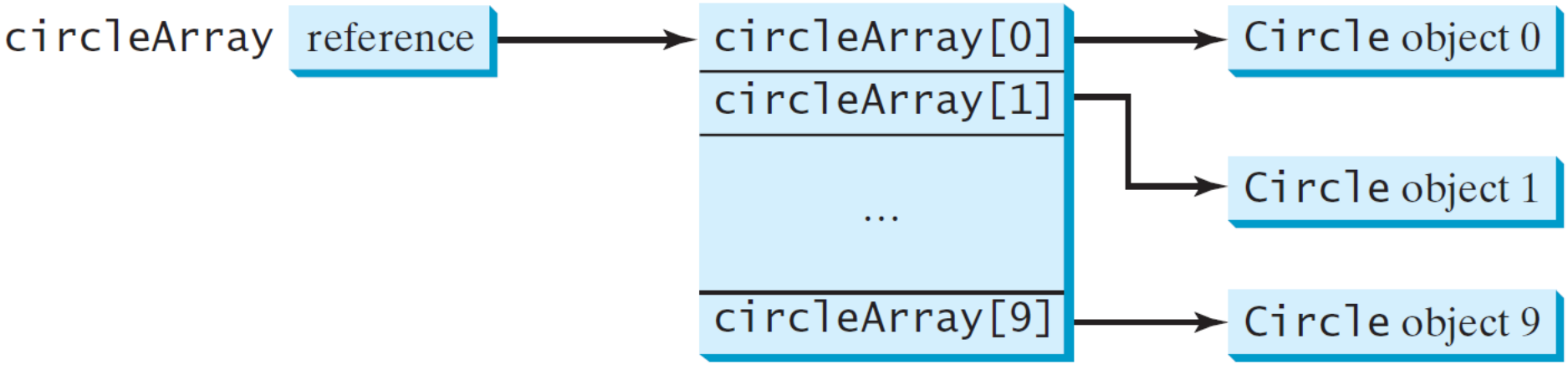
What if we have an array of objects?

```
Circle[] circleArray = new Circle[10];
```

- An array of objects is actually an array of **reference variables**
- So invoking `circleArray[1].getArea()` involves two levels of referencing as shown in the next figure
- `circleArray` references to the entire array. `circleArray[1]` references to a `Circle` object

What if we have an array of objects?

```
Circle[] circleArray = new Circle[10];
```



What if we have an array of objects?

Example: Summarizing the areas of the circles

https://liveexample.pearsoncmg.com/html/TotalArea.html

Array versus **ArrayList**

- Array is a container object that holds a fixed number of values of a **single type**
- The length of an array is established when the array is created
- After creation, its **length is fixed**

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

The `ArrayList` Class

- A very useful class for storing objects
- ArrayList has the generic type `E`
- A concrete type can be specified to replace the **generic type**

Read:
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

```
ArrayList<String> cities = new ArrayList<String>();
```

This ArrayList is used to store **strings**

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E) : void	Appends a new element o at the end of this list.
+add(index: int, o: E) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : E	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : boolean	Removes the element at the specified index.
+set(index: int, o: E) : E	Sets the element at the specified index.

Learn how to work with Java docs:
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

The `ArrayList` Class

- An ArrayList is just an object, and we can create it just like any other object:

```
ArrayList myList = new ArrayList();
//new ArrayList is initially empty
```

- To add an object to the ArrayList, we call the `add()` method provided by the ArrayList class

```
myList.add("New");
myList.add("Item");
```

The `ArrayList` Class

- ArrayList also uses zero-based indexing (take note of the use of `get` method to retrieve an element from the arraylist):

```
System.out.println(list.get(0)); //prints "New"
System.out.println(list.get(1)); //prints "Item"
```

- To get the size (in array, the length) of the arraylist, use the `size()` method

```
int listSize = myList.size();//size() returns 2
```

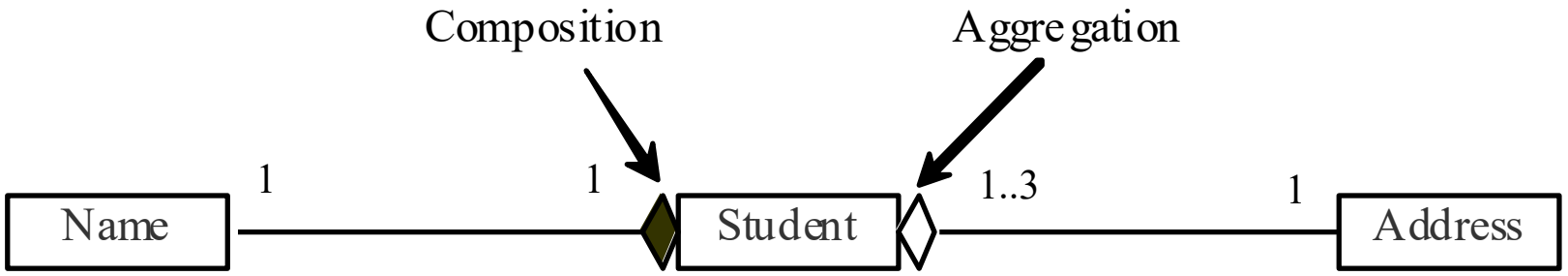
Working with multiple classes

- Classes provide more flexibility and modularity for building reusable software
- We have discussed and seen some of the benefits of developing reusable code using objects and classes
- A program will often comprise of more than one classes

Class Relationships: association, aggregation, composition and inheritance

Working with multiple classes

- Association: a general binary relationship that describes an activity between two classes
- Aggregation: "**has-a**" relationships and represents an ownership relationship between two objects
- Composition: a special case of the aggregation relationship



Working with multiple classes

- An aggregation relationship is usually represented as a data field in the aggregating class

```
public class Name {
   ...
}
```

Aggregated class

```
public class Student {
   private Name name;
   private Address address;

   ...
}
```

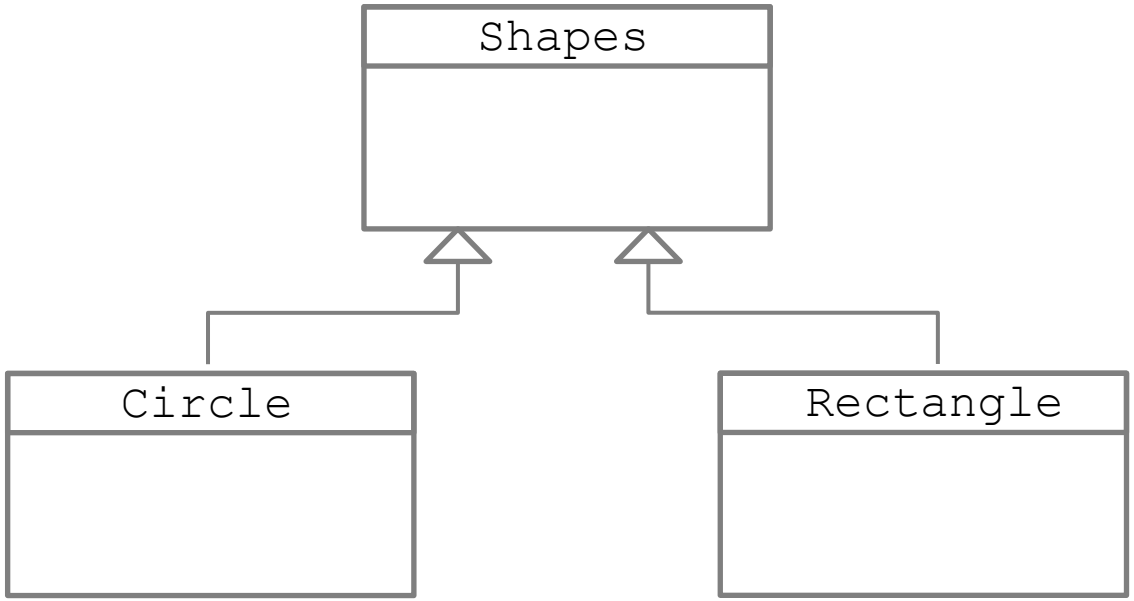
Aggregating class

```
public class Address {
   ...
}
```

Aggregated class

Inheritance

- Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy?



Inheritance

Superclass and subclasses

- **Superclass**: different classes have some common properties and behaviors which can be **generalized in a class** that can be shared by other classes
- A class `C1` extended from another class `C2` is called a **subclass**, and `C2` is called a superclass

The subclass and its superclass are said to form a is-a relationship

INFS1609/2609 Fundamentals of Business Programming

Inheritance

Superclass and subclasses

- The keyword *extends* is used to define the inheritance relationship

```
public class Circle extends GeometricObject
```

INFS1609/2609 Fundamentals of Business Programming

Inheritance

Superclass and subclasses

- A subclass is not exactly a subset of its superclass – in fact a subclass usually contains **more information and methods** than its superclass
- `private` data fields in a superclass are **not accessible** outside the class – you need to use public accessors/mutators

INFS1609/2609 Fundamentals of Business Programming

Superclass and subclasses

Using the `super` keyword

- The keyword `super` refers to the superclass of the class in which `super` appears
- This keyword can be used in two ways:
  - To call a superclass constructor
  - To call a superclass method

Superclass and subclasses

- With `super()`, the superclass no-argument constructor is called
- With `super(parameter list)`, the superclass constructor with a matching parameter list is called
- The call to superclass constructor must be the **first statement** in the constructor

> If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the **no-argument** constructor of the superclass

Superclass and subclasses

- Are superclass's constructor inherited?
  - No, they are not inherited. They are invoked implicitly (the superclass's no-arg constructor is automatically invoked) or explicitly (using the super keyword)

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

INFS1609/2609 Fundamentals of Business Programming

Superclass and subclasses

`GeometricObject` (Superclass)
https://liveexample.pearsoncmg.com/html/SimpleGeometricObject.html

`Circle` (Sub-class)
https://liveexample.pearsoncmg.com/html/CircleFromSimpleGeometricObject.html

`Rectangle` (Sub-class)
https://liveexample.pearsoncmg.com/html/RectangleFromSimpleGeometricObject.html

Constructor Chaining

- Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is known as constructor chaining

When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks

INFS1609/2609 Fundamentals of Business Programming

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee's no-arg constructor

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

Defining a Subclass

A subclass **inherits** from a superclass. You can also:

- Add new properties
- Add new methods
- Override the methods of the superclass

Overriding Methods

- A subclass inherits methods from a superclass
- But sometimes it is necessary for the subclass to **modify the implementation** of a method defined in the superclass
- This is referred to as *method overriding*

Overriding means having two methods with the **same method name and parameters** (i.e., method signature)

Overriding Methods

- An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden

If a method defined in a subclass is `private` in its superclass, the two methods are completely unrelated

Overriding Methods

- Like an instance method, a static method can be inherited
- However, a static method cannot be overridden

If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

The `toString()` method

- The toString() method returns a string representation of the **object**
- The default implementation returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like Loan@15037e5

The `toString()` method

- Very often you would want to override the toString method (e.g. in subclasses) so that it returns a digestible string representation of the object

```java
public class Person {
    private String name;
    private String surname;

    public Person(String n, String s) {
        this.name = n;
        this.surname = s;
    }

    public String getName() {
        return this.name;
    }

    public String getSurname() {
        return this.surname;
    }

    public void setName(String n) {
        this.name = n;
    }

    public void setSurname(String s) {
        this.surname = s;
    }

    @Override
    public String toString() {
        return "Name: " + this.name + " - Surname: " + this.surname;
    }
```

Reflection

What have you learned today?

Thank you!