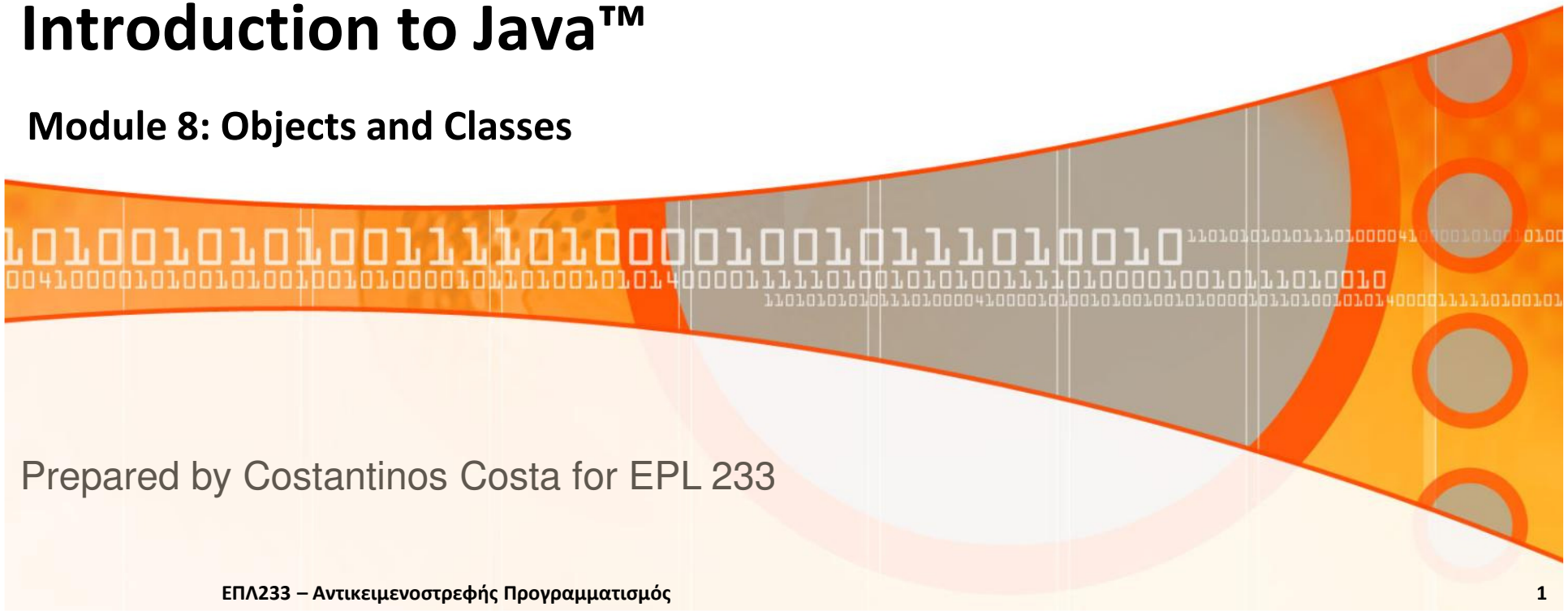
Introduction to Java™

Module 8: Objects and Classes



Prepared by Costantinos Costa for EPL 233

Encapsulation, Polymorphism and Inheritance

- Classic OOP concepts
- Encapsulation: Binding together the properties of an item → creating an object combing of objects
- Polymorphism: Dynamic binding of types
  - An object B which is a subclass of object A can be handled by a type A member field
  - Thus we can call method M of class A from object B
- Inheritance: Refining a base class
  - A new class is derived from the base class
  - The accessible methods and fields of the base class are inherited to the new class
  - Inherited methods can be overridden

Using Objects

- Creating an object
  - Write its class – probably by subclassing
  - Write one (or more) constructor(s)
  - Write its methods
  - Example object:

```
class myObject extends Object {
int a;                      // a member field

myObject() {…}              // default constructor
myObject(int i) {…}         // another constructor

void mehtod1() {…}          // a method that does something

…
}
```

Using Objects

- Initializing an object
  - Create it using the new operator
  - During the creation a constructor is called

```
anObject = new myObject();
```

- Calling an object's methods

```
anObject.method1();
// Calls method1 of the anObject object
```

Defining Methods

- Methods are like functions are in C
- To define a method first we declare what object type it returns (e.g. String toString() {..})
- We can use any of the access modifiers to limit the access to that method (e.g. public String toString() {..})
- We can declare its parameters if any
  (e.g. public String toString(String s) {..})
- Finally we can use any other modifiers we wish: final, abstract, synchronized, static
  (e.g. public static String toString() {..})
  (e.g. public final String toString(String s) {..})

The “this” Keyword

- The “this” keyword can be used to access the methods and member fields of the current object.
- It is a handle of the current object.

```
public class foo {
    int aInt = 2;

    foo() {…}
    public int aMehtod() {…}
    public foo getMe() { return this; }
    public static void main(String[] args){
         this.aInt = this.aMethod();
    }
}
```

Access Modifiers

- "Friendly"
  - No modifier is used - default access limitations
  - Only classes in the same package can use it
  - Cannot be inherited by subclasses in foreign packages
- Public
  - Allows access by everyone
- Private
  - Access is forbidden to everybody except the owner class
- Protected
  - Access limitations are the same as in the "friendly" case
  - Can be inherited by subclasses in foreign packages

Applicable in: classes*, constructors, methods, fields

Subclassing

- **Subclassing involves two classes: the base class and the newly created derived class.**
- **When subclassing we inherit from the base class into the derived class**
- **Creating a subclass:**

```
public class foo extends goo {
int I;

  foo () {
    super();
    I=0;
  }
} // This is a subclass of class goo
```

Overriding methods

- **Methods declared in the subclass as well as the superclass.**
- **When called, the method in the subclass (not the superclass) will be executed.**

Example:

```
class One {
        //constructor
            public void method1() {
             System.out.println("This is class One"); }
            }
class Two extends One {
                        .......
Overriding method →    public void method1() {
                       System.out.println("This is class Two"); }
```

Overriding vs. Overloading

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

class OverridingB {
 public void p(int i) {
 }
}

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

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

class OverloadingB {
 public void p(int i) {
 }
}

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

Interfaces

- The interface keyword takes the abstract concept one step further ("pure" **abstract** class)
- An interface provides only a form, **but no implementation.**
  - It allows the creator to establish the form for a class: method names, argument lists, and return types, but no method bodies.
- Interfaces can contain fields that are implicitly **static** and **final**.
  - Automatically **static** and **final** → cannot be "blank finals"
  - Can be initialized with nonconstant expressions.

Inheritance and Interfaces

- As an interface has no implementation at all many interfaces can being combined (implemented) to form a new derived class
- Valuable when you need to say "An **x** is an **a** *and* a **b** *and* a **c**."
  - In C++ → *multiple inheritance*
    - Carries some rather sticky baggage because each class can have an implementation.
  - In Java → can perform the same act, but only one of the classes can have an implementation.
    - So the problems seen in C++ do not occur with Java when combining multiple interfaces

Inheritance and Interfaces

- As an interface has no implementation at all many interfaces can being combined (implemented) to form a new derived class
- Valuable when you need to say "An **x** is an **a** *and* a **b** *and* a **c**."
  - In C++ → *multiple inheritance*
    - Carries some rather sticky baggage because each class can have an implementation.
  - In Java → can perform the same act, but only one of the classes can have an implementation.
    - So the problems seen in C++ do not occur with Java when combining multiple interfaces

Inheritance example

```
class Animal{
   public void move(){System.out.println("Animals can move");}
}
class Dog extends Animal{
   public void move(){
      super.move(); // invokes the super class method
      System.out.println("Dogs can walk and run");}
}
class SheepDog extends Animal{
   public void move(){
      super.move(); // invokes the super class method
      System.out.println("SheepDogs are protecting sheeps");
   }
}
public class TestDog{
   public static void main(String args[]){
      Animal a1 = new Dog(); // Animal reference but Dog object
      Animal a2 = new SheepDog(); // Animal reference but Dog object
      a1.move();//Runs the method in Dog class
      a2.move();//Runs the method in SheepDog class
   }
}
```

Interface example

```
// Multiple interfaces.
interface CanFight {void fight();}
interface CanSwim {void swim();}
interface CanFly {void fly();}

class ActionCharacter {public void fight() {}}

class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
public void swim() {System.out.println("Hero.swim()");}
public void fly() {System.out.println("Hero.fly()");}
}

public class Adventure {
static void t(CanFight x) {x.fight();}
static void u(CanSwim x) {x.swim();}
static void v(CanFly x) {x.fly();}
static void w(ActionCharacter x) {x.fight();}

public static void main(String[] args) {
Hero h = new Hero();
t(h);// Treat it as a CanFight
u(h);// Treat it as a CanSwim
v(h);// Treat it as a CanFly
w(h);
}
}
```

Task

- Write a program for the well know game “rock-scissor-paper”. The program must have *N* rounds and *M* players. The only time that the players should play is when they have the opposite sex. In each round the program may print the round score. After the *N* rounds the program should choose the winner.

- IMPORTANT: You must use only one ArrayList and the given files(Person.java and Player.java).