

PROGRAMMING III

JAVA LANGUAGE

COURSE 3

PREVIOUS COURSE CONTENT

- ☐ **Classes**
- ☐ **Objects**
- ☐ **Object class**
- ☐ **Access control specifier**
 - ☐ fields
 - ☐ methods
 - ☐ classes
- ☐ **Organizing classes**

COUSE CONTENT

☐ Inheritance

- ☐ Abstract classes

- ☐ Interfaces

- ☐ `instanceof` operator

☐ Nested classes

☐ Enumerations

RELATION BETWEEN CLASSES

☐ What relations between classes exists?

☐ Associations

- ☐ Dependency
- ☐ Association
- ☐ Aggregation
- ☐ Composition

☐ Inheritance

INHERITANCE

- ❑ Inheritance is a mechanism which allows a class A to **inherit members** (data and functions) of a class B. We say “A inherits from B”. Objects of class A thus have access to members of class B without the need to redefine them.
- ❑ Terminology
 - ❑ **Base** class
 - ❑ The class that is inherited
 - ❑ **Derived** class
 - ❑ A specialization of base class
 - ❑ **Kind-of** relation
 - ❑ Class level (Circle is a kind-of Shape)
 - ❑ **Is-a** relation
 - ❑ Object level (The object circle1 is-a shape.)
 - ❑ Types of inheritance
 - ❑ **Simple**
 - ❑ One base class
 - ❑ **Multiple** - NOT SUPPORTED IN JAVA
 - ❑ Multiple base classes

SIMPLE INHERITANCE

❑ Syntax

```
❑ [ClassSpecifier] class ClassName extends BaseClassName  
    { ... }
```

❑ Example

```
public class Figure {  
    Color color;  
    public Figure() {  
        this.color = Color.RED;  
    }  
}  
  
public class Circle extends Figure {  
    int radius;  
    int centerX, centerY;  
    ...  
}
```

❑ A class inherits a single base class

SIMPLE INHERITANCE. CONSTRUCTORS

❑ **super keyword**

- ❑ Reference to the base class

❑ **Example**

```
public class Figure {
    Color color;

    public Figure() {
        this.color = Color.RED;
    }

    public Figure (Color c) {
        this.color = c
    }

    public String toString(){
        return "color: " +
this.color;
    }
}
```

```
public class Circle extends Figure {
    int radius;
    int centerX, centerY;

    public Circle(){
        super();
    }

    public Circle (int r, int x,
                    int y, Color c) {
        super(c);
        this. radius = r;
        this.centerX = x;
        this.centerY = y;
    }

    public String toString() {
        return "["+ this.radius + ", (" +
            this.centerX + "," +
            this.centerY + "), " +
            super.toString() + "]";
    }
}
```

ABSTRACT CLASSES

- ❑ **Abstract classes is a class **declared abstract****

- ❑ It may or not include abstract methods

- ❑ **Abstract method**

- ❑ Method that is only declared **without an implementation**

- ❑ Example

- ❑ `public static void fooMethod(int par1);`

- ❑ **Properties**

- ❑ Abstract classes **cannot** be **instantiated**
 - ❑ Can **contain abstract** and **non abstract** methods
 - ❑ Can **contain fields** that are **not static or final**

INTERFACES

❑ Interfaces

- ❑ similar to class
- ❑ **API** - Application Programming Interfaces
 - ❑ a "contract" that spells out software interactions
- ❑ Can **contain** only
 - ❑ constants
 - ❑ method signature
 - ❑ default methods
 - ❑ static methods
 - ❑ nested types

❑ Syntax

```
[interfaceModifier] interface InterfaceName [implements  
Interface1 [, ..InterfaceN]]{ ... }
```

❑ where

- ❑ interfaceModifier: package, public

INTERFACES

❑ Inheritance

- ❑ a class **can inherit multiple** interfaces
- ❑ An instance method in a subclass with the **same signature** (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method
- ❑ An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a **covariant return type**

❑ Multiple inheritance

- ❑ Multiple inheritance is the ability to inherit method definitions from multiple base (super) classes
- ❑ Java supports **multiple inheritance of type**, which is the ability of a class to implement more than one interface

INTERFACES CAN BE EXTENDED

- ❑ Creation (definition) of interfaces can be done using inheritance
 - ❑ one interface can extend another.
- ❑ Sometimes interfaces are used just as **labeling mechanisms**
 - ❑ Look in the Java API documentation for interfaces like Cloneable or Serializable.
 - ❑ **Optional**
 - ❑ read about Marker design pattern and annotations
- ❑ All interface **methods** are by **default public** so they do not need to be declared public

INTERFACES

❑ Java 1.8

- ❑ Methods with implementation

- ❑ Types

 - ❑ `default methods`

 - ❑ `static methods`

❑ Java 1.9

- ❑ Private methods

- ❑ Private Static methods

INTERFACES. DEFAULT METHODS

- ❑ Enable the add of **new functionalities** to interfaces without breaking the classes that implements that interface

- ❑ Example

```
interface InterfaceA {  
    public void saySomething();  
    default public void sayHi() {  
        System.out.println("Hi");  
    }  
}
```

```
public class MyClass implements InterfaceA {  
  
    @Override  
    public void saySomething() {  
        System.out.println("Hello World");  
    }  
}
```

INTERFACES. DEFAULT METHODS

❑ Conflicts with multiple interfaces

❑ Problem

- ❑ One or more interfaces has a **default** method with the **same signature**

❑ Solution

- ❑ **Provide implementation** for the method in **derived class**
 - ❑ New implementation
 - ❑ Call one of the interfaces implementation

INTERFACES. STATIC METHODS

- ❑ **Similar to default** method except that **can't be override** in subclasses implementation
- ❑ Contain the **complete definition** of the function
- ❑ To **use** a static method, **Interface name** should be instantiated with it
- ❑ **Example**

```
public interface MyData {  
    static boolean isNull(String str) {  
        System.out.println("Interface Null Check");  
        return str == null ? true : "".equals(str) ? true : false;  
    }  
}  
  
public class MyDataImpl implements MyData {  
    public boolean isNull(String str) {  
        System.out.println("Impl Null Check");  
        return str == null ? true : false;  
    }  
    public static void main(String args[]){  
        MyDataImpl obj = new MyDataImpl();  
        obj.isNull("abc");  
    }  
}
```

What is the result of the program?

- a) Interface Null Check
- b) Impl Null Check

Answer

a)

INTERFACES. PRIVATE METHODS

- ❑ No need to write duplicate code, hence more code reusability.
- ❑ Expose only intended methods implementations to clients.
- ❑ Example

```
public interface MyLogging{  
    default void infoLog(String msg){  
        log("INFO", msg);  
    }  
    default void infoErr(String msg){  
        log("Error", msg);  
    }  
    private void log(String prefix, String msg){  
        // write into a database or file  
    }  
    // other abstract methods  
}
```

← The class that uses the logging interface does not have to create an instance of the MyLogging object

FUNCTIONAL INTERFACES

- ❑ An interface with **exactly one** abstract method is known as **Functional Interface**
 - ❑ annotation `@FunctionalInterface` mark an interface as Functional Interface
 - ❑ lambda expressions

CASTING OBJECTS

- ❑ A object of a **derived** class can be **cast as** an object of the **base** class
- ❑ When a method is called, the **selection** of which version of **method** is run is totally **dynamic**
 - ❑ overridden methods are dynamic

POLYMORPHISM

- ❑ A reference can be polymorphic, which can be defined as **"having many forms"**
 - ❑ `obj.dolt();`
 - ❑ This line of code might execute different methods at different times if the object that `obj` points to changes
- ❑ **Polymorphic references** are resolved at **run time**; this is called **dynamic binding**
- ❑ Careful use of polymorphic references can lead to **elegant, robust software designs**
- ❑ **Polymorphism** can be accomplished using **inheritance** or using **interfaces**

INSTANCEOF

- ❑ Knowing the **type of an object** during run time
- ❑ Usage
 - ❑ `object instanceof type`
- ❑ It can be very useful when writing generalized routines that operate on objects of a complex class hierarchy
- ❑ It will cause a compiler error if the comparison is done with objects which are not in the same class hierarchy.
- ❑ Returns true if the type could be cast to the reference type without causing a `ClassCastException`, otherwise it is false.

NESTED CLASSES

- ❑ Define a **class within** another **class**.
- ❑ Why use nested classes?
 - ❑ It is a way of **logically grouping** classes that are only used in one place
 - ❑ It increases **encapsulation**
 - ❑ It can lead to more **readable** and **maintainable code**
- ❑ Types
 - ❑ Static member classes
 - ❑ Member classes
 - ❑ Local classes
 - ❑ Anonymous classes

NESTED CLASSES

❑ Types

❑ Static member classes

- ❑ is a static member of a class
- ❑ a static member class has **access** to **all static methods** of the parent, or top-level, class.

❑ Member classes

- ❑ is also defined as a member of a class
- ❑ is instance specific and has **access** to any and **all methods and members**, even the parent's `this` reference

❑ Local classes

- ❑ are declared within a **block of code** and are visible only within that block

❑ Anonymous classes

- ❑ is a local class that has **no name**

NESTED CLASSES

❑ Example

```
public class Outer{  
    private class Inner  
    {  
        // inner class instance variables  
        // inner class methods  
  
    } // end of inner class definition  
  
    // outer class instance variables  
    // outer class methods  
}
```

PUBLIC INNER CLASSES

- ❑ If an inner class is **marked public**, then it can be **used outside** of the outer class
- ❑ In the case of a **nonstatic** inner class, it must be created using an object of the outer class

```
BankAccount account = new BankAccount();  
BankAccount.Money amount = account.new Money("41.99");
```

- ❑ Note that the prefix **account.** must come before **new**
- ❑ The new object **amount** can now invoke methods from the inner class, but only from the inner class

PUBLIC INNER CLASSES

- ❑ In the case of a **static** inner class, the procedure is similar to, but simpler than, that for nonstatic inner classes

```
OuterClass.InnerClass innerObject =  
    new OuterClass.InnerClass();
```

- ❑ Note that all of the following are acceptable

```
innerObject.nonstaticMethod();  
innerObject.staticMethod();  
OuterClass.InnerClass.staticMethod();
```

INNER CLASS AND INHERITANCE

- ❑ **Given an OuterClass that has an InnerClass**
 - ❑ Any DerivedClass of OuterClass will automatically have InnerClass as an inner class
 - ❑ In this case, the DerivedClass **cannot override** the **InnerClass**
- ❑ An **outer class** can be a **derived** class
- ❑ An **inner class** can be a **derived** class

ANONYMOUS CLASSES

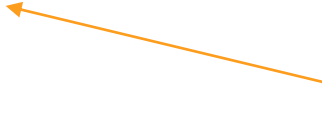
- ❑ If an object is to be created, but there is **no need to name** the object's class, then an anonymous class definition can be used
 - ❑ The class definition is **embedded inside the expression** with the new operator
 - ❑ An anonymous class is an abbreviated notation for creating a simple local object "in-line" within any expression, simply by wrapping the desired code in a "new" expression.
- ❑ **Anonymous classes are sometimes used when they are to be assigned to a variable of another type**
 - ❑ The other type must be such that an object of the anonymous class is also an object of the other type
 - ❑ The other type is usually a Java interface

ANONYMOUS CLASSES

❑ Example

```
interface Foo {  
    void doSomething();  
}  
public class Test {  
    public static void main (String args[]) {  
        Foo obj = new Foo() {  
            void doSomething() {  
                System.out.println("test");  
            }  
        };  
        obj.doSomething();  
    }  
}
```

Anonymous Class



ENUMERATIONS

- ❑ Enumerated values are used to represent a set of **named values**

- ❑ These were often stored as **constants**.

- ❑ **For example**

```
public static final int SUIT_CLUBS = 0;  
public static final int SUIT_DIAMONDS = 1;  
public static final int SUIT_HEARTS = 2;  
public static final int SUIT_SPADES = 3;
```

ENUMERATIONS

❑ Issues with previous approach

❑ Acceptable values are not obvious

- ❑ Since the values are just integers, it's hard at a glance to tell what the possible values are.

❑ No type safety

- ❑ Since the values are just integers, the compiler will let you substitute any valid integer

❑ No name-spacing

- ❑ With our card example, we prefixed each of the suits with "SUIT_" .
- ❑ We chose to prefix all of those constants with this prefix to potentially disambiguate from other enumerated values of the same class.

❑ Not printable

- ❑ Since they are just integers, if we were to print out the values, they'd simply display their numerical value.

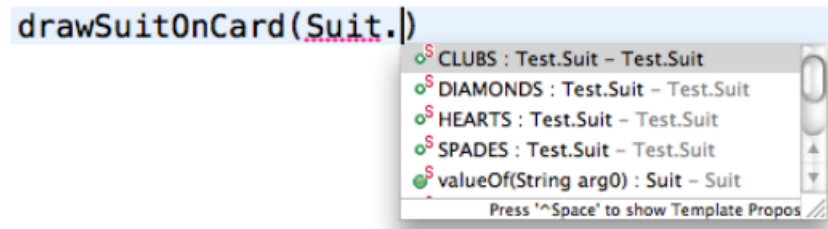
ENUMERATIONS

- ❑ **Java 5** added an `enum` type to the language
- ❑ Declared using the `enum` keyword instead of `class`
- ❑ **Simplest form**, contains a comma separated list of names representing each of the possible options.

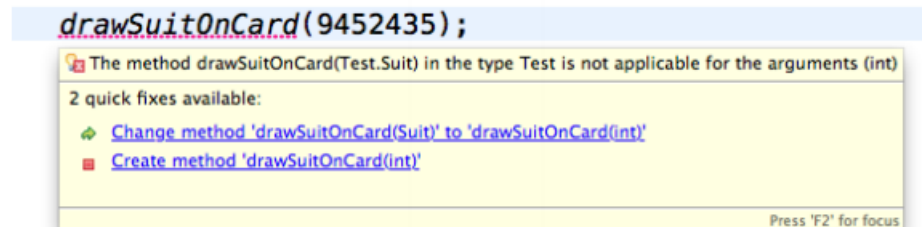
```
public enum Suit { CLUBS, DIAMONDS, HEARTS,  
    SPADES }
```

ENUMERATIONS

- ❑ Acceptable values are now obvious — must choose one of the Suit enumerated values...

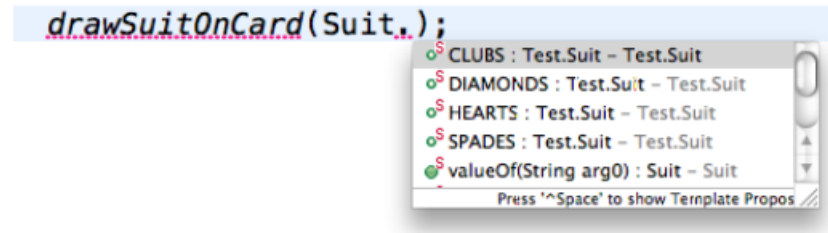


Type safety — possible values are enforced by the compiler



ENUMERATIONS

- ❑ Every value is name-spaced off of the enum type itself.



- ❑ Printing the enum value is actually readable.

```
System.out.print("Card is a Queen of " + Suit.HEARTS);
```

ENUMERATIONS

☐ **Additional Benefits**

- ☐ Storage of additional information
- ☐ Retrieval of all enumerated values of a type
- ☐ Comparison of enumerated values

ENUMERATIONS.

ADDITIONAL BENEFITS

❑ Enums are objects

- ❑ So they can have...
 - ❑ Member variables
 - ❑ Methods

❑ For example...

- ❑ Embed the color of the suit within the Suit.
- ❑ Read the value using a getter, etc.

```
public enum Suit {  
    CLUBS (Color.BLACK),  
    DIAMONDS (Color.RED),  
    HEARTS (Color.RED),  
    SPADES (Color.BLACK);  
    private Color color;
```

```
    Suit(Color c) {  
        this.color = c;
```

```
    }  
    public Color getColor() {  
        return this.color;
```

```
    }
```

Constructor, add
supplementary
information

Method to access
supplementary
information

How to use?

```
public static void main(String[] args) {  
    Suit s = Suit.CLUBS;  
    System.out.println("Card color: " + s.getColor());  
}
```

ENUMERATIONS.

ADDITIONAL BENEFITS

RETRIEVAL OF ALL ENUMERATED VALUES

- ❑ All enum types will automatically have a `values()` method that returns an array of all enumerated values for that type.

```
Suit[ ] suits =  
Suit.values();  
for(Suit s : suits) {  
    System.out.println(s  
);  
}
```

COMPARISON OF ENUMERATED VALUES

- ❑ It is possible to compare enums using the `==` operator.

```
if(suit == Suit.CLUBS) {  
    // do something  
}
```

- ❑ can also be used with the switch control structure

```
Suit suit = /* ... */;  
switch (suit) {  
    case CLUBS:  
    case SPADES:  
        // do something  
        break;  
    case HEARTS:  
    case DIAMONDS:  
        // do something else  
        break;  
    default:  
        // yet another thing  
        break;  
}
```