Objectives

- In this session, you will learn to:
  - Create final classes, methods, and variables
  - Create and use enumerated types
  - Use the static import statement
  - Create abstract classes and methods
  - Create and use an interface
  - Define exceptions
  - Use try, catch, and finally statements
  - Describe exception categories
  - Identify common exceptions
  - Develop programs to handle your own exceptions
  - Use assertions
  - Distinguish appropriate and inappropriate uses of assertions
  - Enable assertions at runtime

The final Keyword

- The `final` keyword is used for security reasons.
- It is used to create classes that serve as a standard.
- It implements the following restrictions:
  - You cannot subclass a `final` class.
  - You cannot override a `final` method.
  - A `final` variable is a constant.
  - All methods and data members in a `final` class are implicitly final.
- You can set a `final` variable once only, but that assignment can occur independently of the declaration; this is called a blank final variable.

Blank Final Variables

- A `final` variable that is not initialized in its declaration; its initialization is delayed:
  - A blank final instance variable must be assigned in a constructor.
  - A blank final local variable can be set at any time in the body of the method.
- It can be set once only.

Enumerated Types

- An `enum` type field consist of a fixed set of constants.
- You can define an `enum` type by using the `enum` keyword. For example, you would specify a days-of-the-week `enum` type as:

```
public enum Day { SUNDAY, MONDAY, TUESDAY,
WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
```

- The `enum` class body can include methods and other fields.
- The compiler automatically adds some special methods when it creates an `enum`.
- All `enums` implicitly extend from `java.lang.Enum`. Since Java does not support multiple inheritance, an `enum` cannot extend anything else.

Static Imports

- Imports the static members from a class:

```
import static
<pkg_list>.<class_name>.<member_name>;
```

OR

```
import static <pkg_list>.<class_name>.*;
```

- Imports members individually or collectively:

```
import static cards.domain.Suit.SPADES;
```

OR

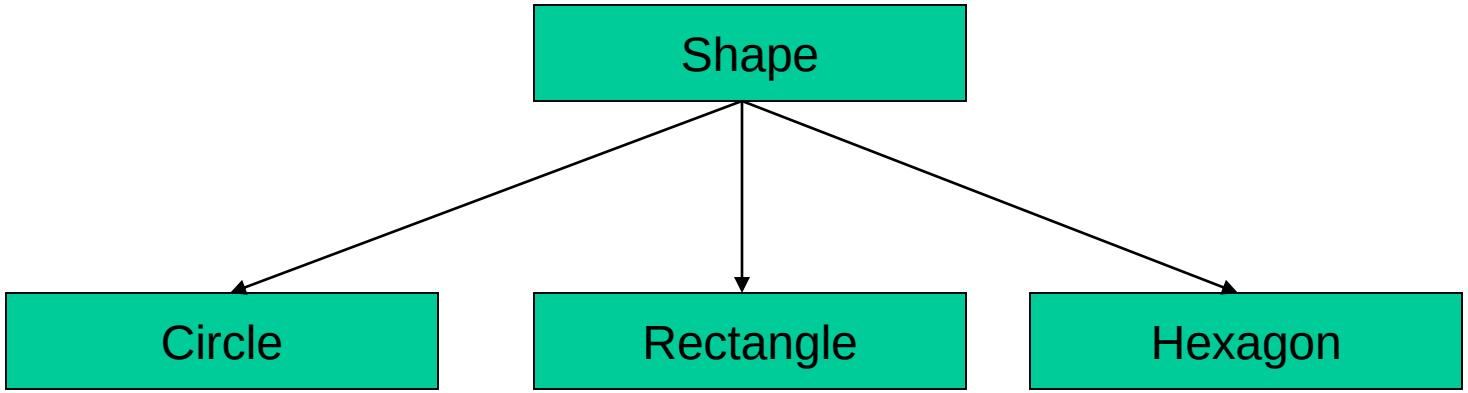
```
import static cards.domain.Suit.*;
```

- There is no need to qualify the static constants:

```
PlayingCard card1 = new PlayingCard(SPADES,
2);
```

Abstract Classes

- An abstract class is declared with `abstract` access specifier and it may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed. For example:



Abstract Classes (Contd.)

- An abstract class defines the common properties and behaviors of other classes.
- It is used as a base class to derive specific classes of the same type. For example:

```
abstract class Shape
{
public abstract float calculateArea();
}
```

- The preceding abstract method, calculateArea, is inherited by the subclasses of the Shape class. The subclasses Rectangle, Circle, and Hexagon implement this method in different ways.

Abstract Classes (Contd.)

- A simple example of implementation of Abstract Method:

```
public class Circle extends Shape
{
    float radius;
    public float calculateArea()
    {
    return ((radius * radius)* (22/7));
    }
}
```

- In the preceding example, the calculateArea() method has been overridden in the Circle class.

Interfaces

- A public interface is a contract between client code and the class that implements that interface.
- A Java interface is a formal declaration of such a contract in which all methods contain no implementation.
- Many unrelated classes can implement the same interface.
- A class can implement many unrelated interfaces.
- Syntax of a Java class declaration with interface implementation is as follows:

```
<modifier> class <name> [extends
<superclass>]
[implements <interface> [,<interface>]* ]
{   <member_declaration>*
}
```

Interfaces (Contd.)

- Interfaces are used to define a behavior protocol (standard behavior) that can be implemented by any class anywhere in the class hierarchy. For example:

  Consider the devices TV and VDU. Both of them require a common functionality as far as brightness control is concerned. This functionality can be provided by implementing an interface called BrightnessControl which is applicable for both the devices.
- Interfaces can be implemented by classes that are not related to one another.
- Abstract classes are used only when there is a kind-of relationship between the classes.

Interfaces (Contd.)

- Uses of Interfaces:
  - Declaring methods that one or more classes are expected to implement
  - Determining an object's programming interface without revealing the actual body of the class
  - Capturing similarities between unrelated classes without forcing a class relationship
  - Simulating multiple inheritance by declaring a class that implements several interfaces