IT602: Object-Oriented Programming

Lecture - 14

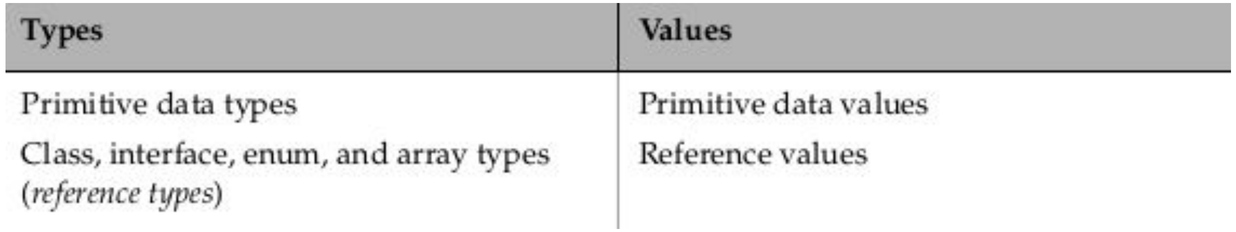
Arrays and Subtypes + More OOP Concepts

Arpit Rana

31st March 2022

Arrays and Subtypes

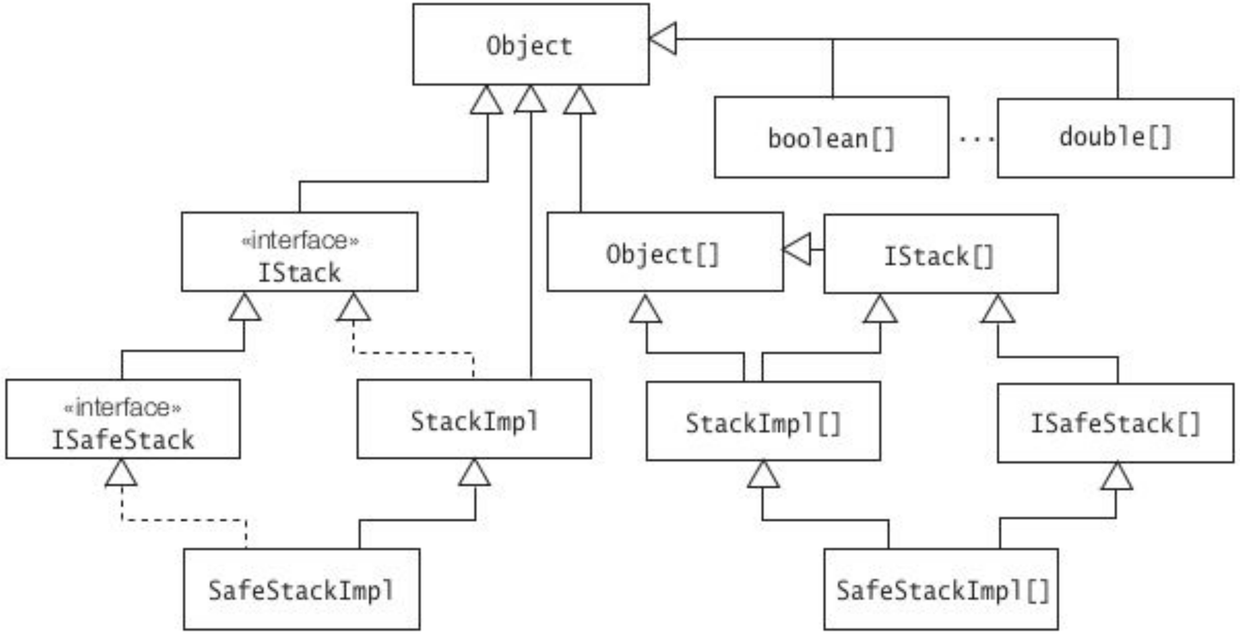
Only primitive data and reference values can be stored in variables. Only class and array types can be explicitly instantiated to create objects.



Types	Values
Primitive data types	Primitive data values
Class, interface, enum, and array types (*reference types*)	Reference values

Arrays and Subtype Covariance

Arrays are objects in Java. Array types (`boolean[]`,`Object[]`, `StackImpl[]`) implicitly augment the inheritance hierarchy.

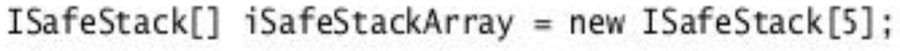


Arrays and Subtype Covariance

- All reference types are subtypes of the `Object` type. This applies to classes, interfaces, enum, and array types, as these are all reference types.
- All arrays of reference types are also subtypes of the array type `Object[]`, but arrays of primitive data types are not. Note that the array type `Object[]` is also a subtype of the `Object` type.
- There is no subtype-supertype relationship between a type and its corresponding array type.

Arrays and Subtype Covariance

We can create an array of an interface type, but we cannot instantiate an interface (as is the case with *abstract classes*).



```
ISafeStack[] iSafeStackArray = new ISafeStack[5];
```

The array object can accommodate five references of the type `ISafeStack`. But, they are initialized to the default value `null`.

Array Store Check

The following assignment is valid, as a supertype reference (`StackImpl[]`) can refer to objects of its subtype (`SafeStackImpl[]`):

```
StackImpl[] stackImplArray = new SafeStackImpl[2];      // (1)
```

Since `StackImpl` is a supertype of `SafeStackImpl`, the following assignment is also valid:

```
stackImplArray[0] = new SafeStackImpl(10);               // (2)
```

Since the type of `stackImplArray[i]` for (1<=i<2), is `StackImpl`, it should be possible to do the following assignment as well:

```
stackImplArray[1] = new StackImpl(20);          // (3) ArrayStoreException
```

The assignment at (3) results in an `ArrayStoreException` to be thrown at runtime, as a `SafeStackImpl[]` object cannot possibly contain objects of type `StackImpl`.

Reference Values and Conversion

Reference values, like primitive values, can be assigned, cast, and passed as arguments. Conversions can occur in the following contexts:

- assignment
- method invocation
- casting

Conversions that are from a *subtype* to its *supertypes* are allowed, other conversions require an explicit cast or are otherwise illegal.

Reference Value Assignment Conversion

In the context of assignments, the following conversions are permitted:

- widening primitive and reference conversions (long ← int, Object ← String)
- boxing conversion of primitive values, followed by optional widening reference conversion (Integer ← int, Number ← Integer ← int)
- unboxing conversion of a primitive value wrapper object, followed by optional widening primitive conversion (long ← int ← Integer)

And only for assignment conversions, we have the following:

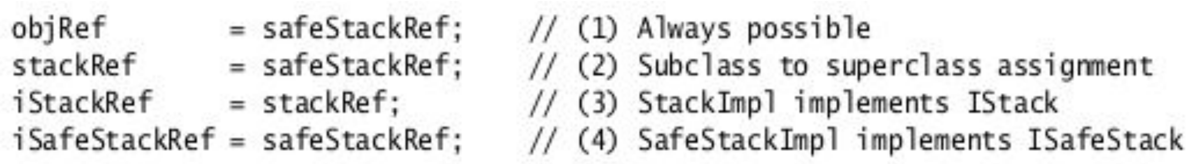
- narrowing conversion for constant expressions of non- long integer type, with optional boxing (Byte ← byte ← int)

Note that the above rules imply that a widening conversion cannot be followed by any boxing conversion, but the converse is permitted.

Reference Value Assignment Conversion

In the context of assignments, the following conversions are permitted:

- If the SourceType is a class type, the reference value in srcRef may be assigned to the destRef reference, provided the DestinationType is one of the following:
  - DestinationType is a superclass of the subclass SourceType .
  - DestinationType is an interface type that is implemented by the class SourceType .

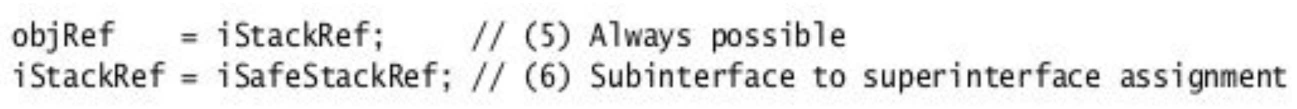


```
objRef        = safeStackRef;     // (1) Always possible
stackRef      = safeStackRef;     // (2) Subclass to superclass assignment
iStackRef     = stackRef;         // (3) StackImpl implements IStack
iSafeStackRef = safeStackRef;     // (4) SafeStackImpl implements ISafeStack
```

Reference Value Assignment Conversion

In the context of assignments, the following conversions are permitted:

- If the SourceType is an interface type, the reference value in srcRef may be assigned to the destRef reference, provided the DestinationType is one of the following:
  - DestinationType is Object .
  - DestinationType is a superinterface of subinterface SourceType .

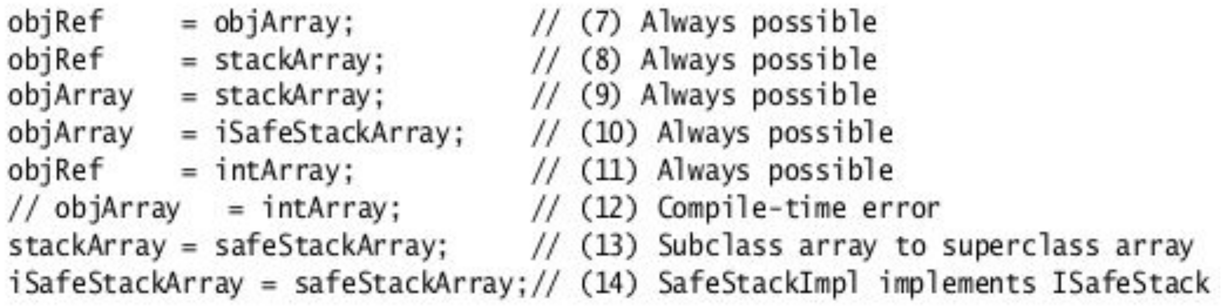


```
objRef    = iStackRef;     // (5) Always possible
iStackRef = iSafeStackRef; // (6) Subinterface to superinterface assignment
```

Reference Value Assignment Conversion

In the context of assignments, the following conversions are permitted:

- If the SourceType is an array type, the reference value in srcRef may be assigned to the destRef reference, provided the DestinationType is one of the following:
  - DestinationType is Object
  - DestinationType is an array type, where the element type of the SourceType is assignable to the element type of the DestinationType



```
objRef      = objArray;          // (7) Always possible
objRef      = stackArray;        // (8) Always possible
objArray    = stackArray;        // (9) Always possible
objArray    = iSafeStackArray;   // (10) Always possible
objRef      = intArray;          // (11) Always possible
// objArray   = intArray;        // (12) Compile-time error
stackArray = safeStackArray;     // (13) Subclass array to superclass array
iSafeStackArray = safeStackArray;// (14) SafeStackImpl implements ISafeStack
```

IT602: Object-Oriented Programming

More OOP Concepts

Polymorphism

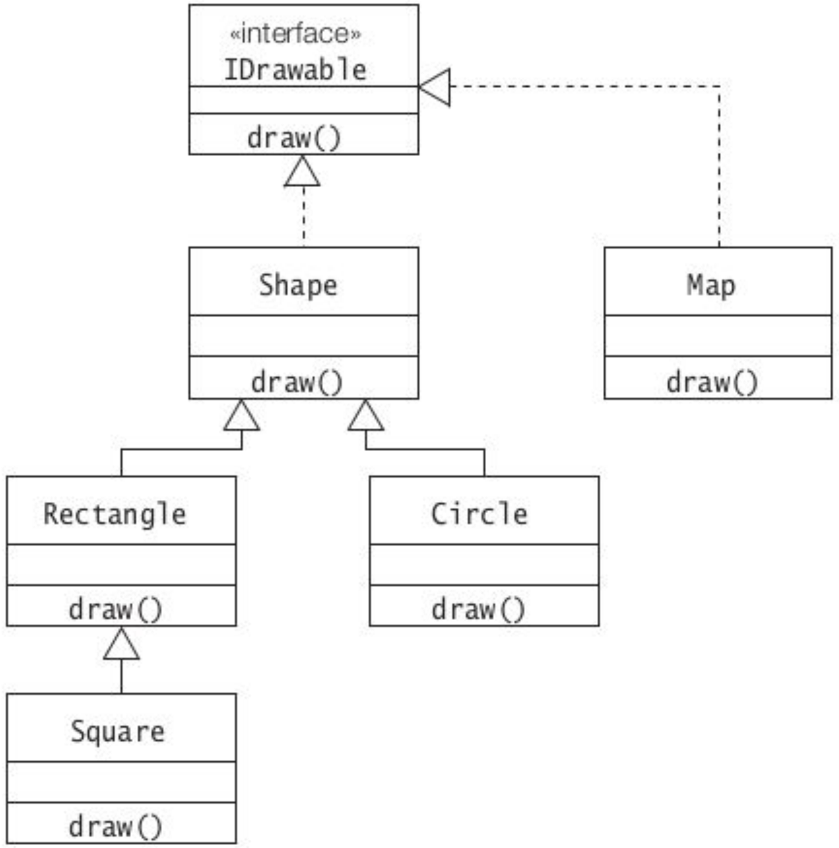
Which object a reference will actually denote during runtime cannot always be determined at compile time.

- ***Polymorphism*** allows a reference to denote objects of different types at different times during execution.
    - A *supertype* reference exhibits polymorphic behavior since it can denote objects of its *subtypes*.
- Polymorphism is achieved through inheritance and interface implementation.
- Code relying on polymorphic behavior will still work without any change if new subclasses or new classes implementing the interface are added.

Dynamic Method Lookup

- ***Dynamic method lookup*** is the process of determining which method definition a method signature denotes during runtime, based on the type of the object.
- When a non-private instance method is invoked on an object, the method definition actually executed is determined by -
  - the type of the object at runtime and
  - the method signature.
- A call to a private instance method is not polymorphic.
  - Such a call can only occur within the class and gets bound to the private method implementation at compile time.

Example



«interface»
IDrawable
draw()
Shape
draw()
Map
draw()
Rectangle
draw()
Circle
draw()
Square
draw()

Example

```java
interface IDrawable {
  void draw();
}

class Shape implements IDrawable {
  public void draw() { System.out.println("Drawing a Shape."); }
}

class Circle extends Shape {
  public void draw() { System.out.println("Drawing a Circle."); }
}

class Rectangle extends Shape {
  public void draw() { System.out.println("Drawing a Rectangle."); }
}

class Square extends Rectangle {
  public void draw() { System.out.println("Drawing a Square."); }
}

class Map implements IDrawable {
  public void draw() { System.out.println("Drawing a Map."); }
}

public class PolymorphRefs {
  public static void main(String[] args) {
    Shape[] shapes = {new Circle(), new Rectangle(), new Square()};     // (1)
    IDrawable[] drawables = {new Shape(), new Rectangle(), new Map()};  // (2)

    System.out.println("Draw shapes:");
    for (Shape shape : shapes)                                          // (3)
      shape.draw();

    System.out.println("Draw drawables:");
    for (IDrawable drawable : drawables)                                // (4)
      drawable.draw();
  }
}
```

Output of the Program

```
Draw shapes:
Drawing a Circle.
Drawing a Rectangle.
Drawing a Square.
Draw drawables:
Drawing a Shape.
Drawing a Rectangle.
Drawing a Map.
```

Basic Concepts in Object-Oriented Design

Object-Oriented Design (OOD) is one approach to software design to achieve maintainability, reusability, extensibility, and reliability.

OOD is the process of planning a system of interacting objects for the purpose of solving a software problem.

- Detailed discussion on OOD is out of the scope of this course.

Encapsulation

An object has *properties* and *behaviors* that are encapsulated inside the object.

- Only the contract (services) defined by the object is available to the clients.
- The implementation of its properties and behavior is not a concern of the clients.

Encapsulation helps to make clear the distinction between an object's contract and implementation.

- The implementation of an object can change without implications for the clients.

Encapsulation

In Java encapsulation is achieved through information hiding at different levels of granularity:

- *method or block level*

  Localizing information in a method hides it from the outside.

- *class level*

  The accessibility of members declared in a class can be controlled through member accessibility modifiers.

- *package level*

  Classes that belong together can be grouped into relevant packages by using the package statement.

Cohesion

Cohesion is an inter-class measure of how well-structured and closely-related the functionality is in a class.

The objective is to design classes with high cohesion, that perform well-defined and related tasks (also called *functional cohesion*).

- The public methods of a highly cohesive class typically implement a single specific task that is related to the purpose of the class.
- A method in one class should not perform a task that should actually be implemented by one of the other two classes.
    - In an MVC-based application, the respective classes for the Model, the View, and the Controller should be focused on providing functionality that only relates to their individual purpose.

Cohesion

Cohesion is an inter-class measure of how well-structured and closely-related the functionality is in a class.

- Lack of cohesion in a class means that the purpose of the class is not focused (also called *coincidental cohesion*)
- This may eventually impact the maintainability of the application.

Coupling

Coupling is a measure of intra-class dependencies.

- Objects need to interact with each other, therefore dependencies between classes are inherent in OO design.
- These dependencies should be minimized in order to achieve loose coupling, which aids in creating extensible applications.

High cohesion and loose coupling help to achieve the main goals of OO design.

Next lecture -

Files and Streams