Object-Oriented Programming

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

HW6

- Last assignment
- Due on the last day of class (Dec. 9) NOON (**no late submission**)

Abstract Data Types

Primitive types: values and operation on values

User-defined types: records, lists, …

Focus on values

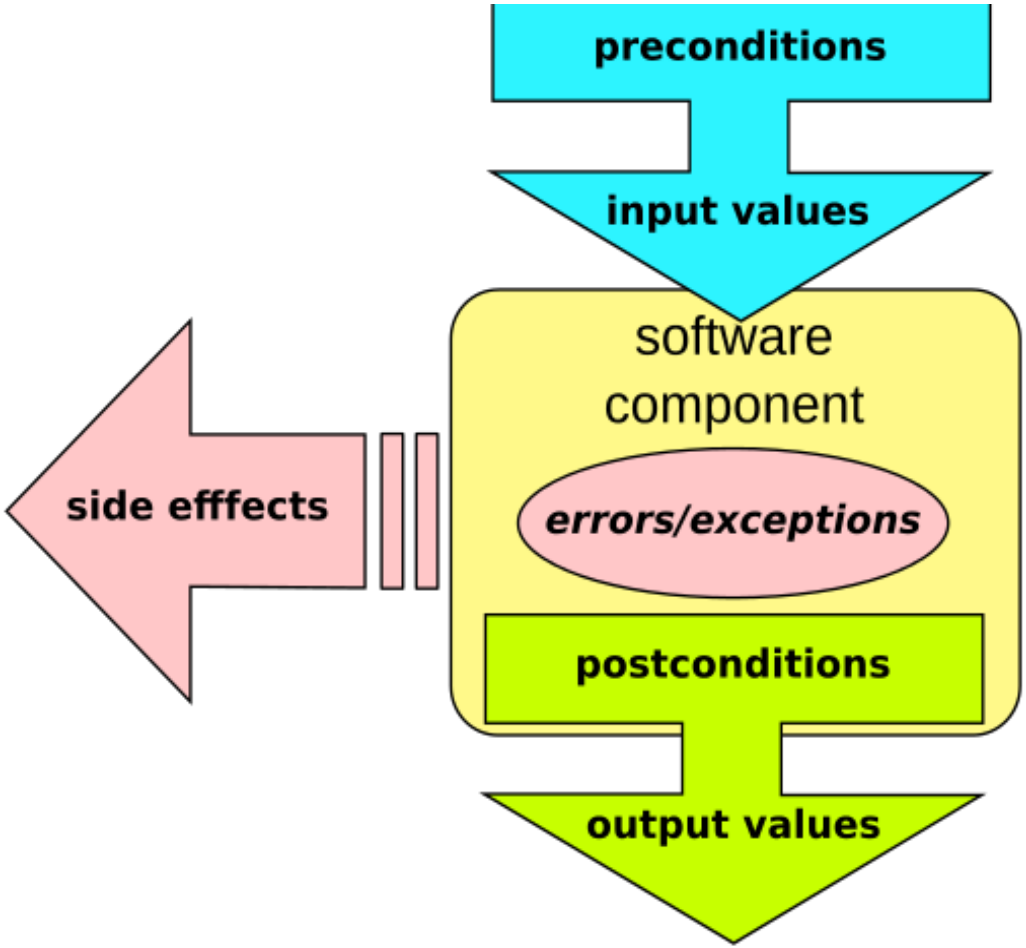
ADT: defined by a set of operations on a type

Focus on operation

Stack is a type with `new, pop, push, empty` …

Design by Contract

A good ADT should define a contract:



ADT makes programs more modular, but

- No simple way to extend existing code
- No subtype polymorphism

Object-Oriented Programming

OOP Terminology

Class: *a richer version of* ADT

Object (Instance): a variable of a class (a *value* of a *type*)

Field: variable in a class

Method: operation in a class

Object-Oriented Programming

Key elements:

- Encapsulation
- Subtyping
- Inheritance

Encapsulation (Information Hiding)

- Group data and operations in one place
- Hide irrelevant details

Visibility Modifiers (Java)

Apply to fields and methods

Modifier	Significance	Comments
public	Accessible everywhere	Normally not used for fields
private	Accessible only within class	May limit extensibility
protected	Accessible from subclasses (and other classes in same package)	
(no modifier)	other classes in same package	

Getters and Setters

Exposing inner state normally breaks modularity

```
public class Rational {
    public int p,q; // represents p/q
    // class invariant: q > 0, gcd(p,q) = 1
    //                    Note: gcd(0,x) = x
```

Breaks module Rational

```
Rational a = new Rational(2,3);
a.q = 0;
a.add(new Ratoinal(1,3));
```

Getters and Setters

Inner state normally exposed by getters and setters

```
public class Rational {
  private int p,q; // represents p/q
  public int getQ() {…};
  /** Requires: nq!=0 */
  public void setQ(int nq) {assert (nq!=0); …};
```

Breaks contract

```
Rational a = new Rational(2,3);
setQ(0);
a.add(new Ratoinal(1,3));
```

Singleton Pattern (Private Constructor)

A class with one single instance?

- A resource manager
- A class that represents empty linked lists

```
class ResManager {
    private ResManager() {…}
    private ResManager manager=null;
    public getInstance() {
        if (manager == null)
            manager = new ResManager();
        return manager;
    }
}
```

Names and Packages

Classes in Java live in **packages**

E.g., String is the shorthand for java.lang.String

java.lang is the package containing class String

Fully qualified name is needed for names in other packages, or import that package

```
import java.util.ArrayList;
```

Object-Oriented Programming

Key elements:

- Encapsulation
- ***Subtyping***
- Inheritance

Interface

A modular mechanism that allows programmers to create interfaces for classes

An “abstract” impl. that specifies public methods

```
interface Shape {
    public double area();
    public int edges();
}
```

Implementation

An interface may have multiple implementations

```
class Rectangle implements Shape {
  double width, length;
  public double area() {return width*length};
  public int edges() {return 4};
}
```

```
class Circle implements Shape {
  double radius;
  public double area() {return 3.14*radius*radius};
  public int edges() {return 1};
}
```

Implementation

```
interface Shape {
  public double area();
  public int edges();
}
```

```
class Rectangle implements Shape {
  double width, length;
  public double area() {return width*length};
  public int edges() {return 4};
}
```

- Compiler checks that each method declared in interface is implemented with same signature
- Implementation may introduce new methods (e.g., constructor)

Use of Interface Types

We cannot create new instance of an interface

```
Rectangle r = new Rectangle(2,3); // OK
Circle c = new Circle(3); // OK
Shape s = new Shape(); // illegal
```

Implementation
is not complete

Use of Interface Types

But, we can write polymorphic function using interface types:

Implementation must have this method

```
void foo (Shape s) { s.area(); }
Rectangle r = new Rectangle(2,3);
Circle c = new Circle(3);
foo(r);
foo(c);
```

Subtyping Polymorphism

τ_1 is a subtype of τ_2 (written $\tau_1 \leq \tau_2$) if a program can use a value of type τ_1 whenever it would use a value of type τ_2.

```
void foo (Shape s) { s.area(); }
Rectangle r = new Rectangle(2,3);
Circle c = new Circle(3);
foo(r);
foo(c);
```

Subtyping

Sometimes, every value of type B is of type A

- e.g., a `Rectangle` is always a `Shape`

We say B is a *subtype* of A, meaning

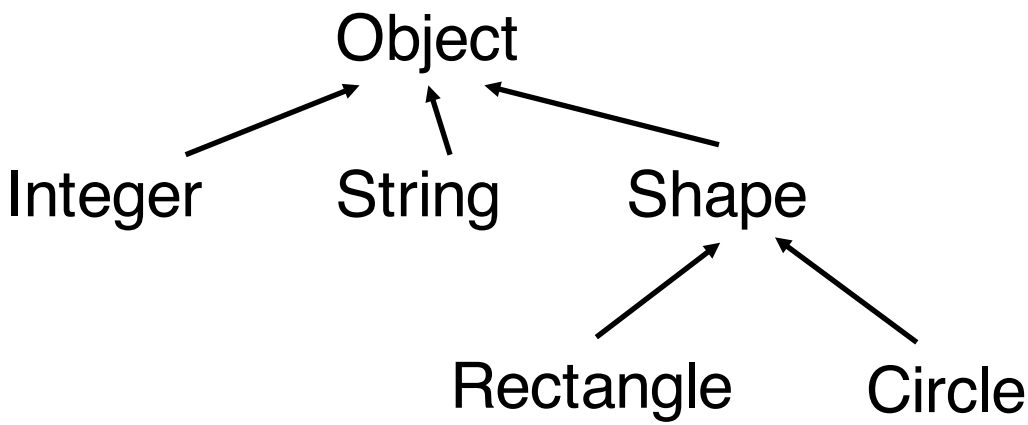
"every object that satisfies interface of B also satisfies interface of A”

i.e., ADT B has all operations in ADT A

Subtyping Relation

We write A $\leq$ B to mean A is a subtype of B

Relation $\leq$ defines a partial ordering on types



Reflexibility: $\forall T . T \leq T$

Antisymmetry: $\forall T_1, T_2 . T_1 \leq T_2 \wedge T_2 \leq T_1 \Rightarrow T_1 = T_2$

Transitivity: $\forall T_1, T_2, T_3 . T_1 \leq T_2 \wedge T_2 \leq T_3 \Rightarrow T_1 \leq T_3$

Casting

Change type at run time

- Upcast: change to a supertype
- Downcast: change to a subtype

```
Rectangle r = new Rectangle(2,3); // OK
Shape s = r; // Upcast is always safe
r = (Rectangle) s; // OK, but downcast
                   // is not always safe
```

Java checks type casts at run time

Type Test

Type of a value can be tested at run time:

```
Shape s = …;
if (s instance of Rectangle) {
  r = (Rectangle) s; // Downcast is safe here
}
```

Object-Oriented Programming

Key elements:

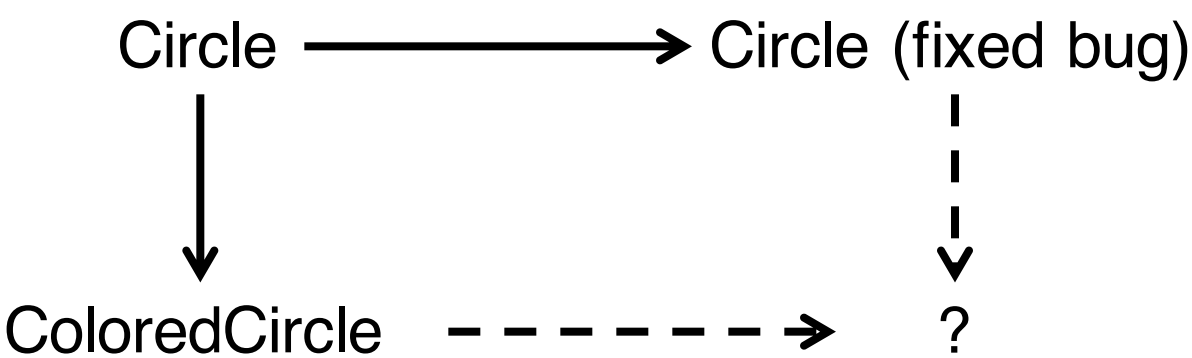
- Encapsulation
- Subtyping
- ***Inheritance***

Motivating Example

```
class Circle implements Shape {
  double radius;
  public double area() {return 3.14*radius*radius};
  public int edges() {return 1};
}
```

How to implement `ColoredCircle`?

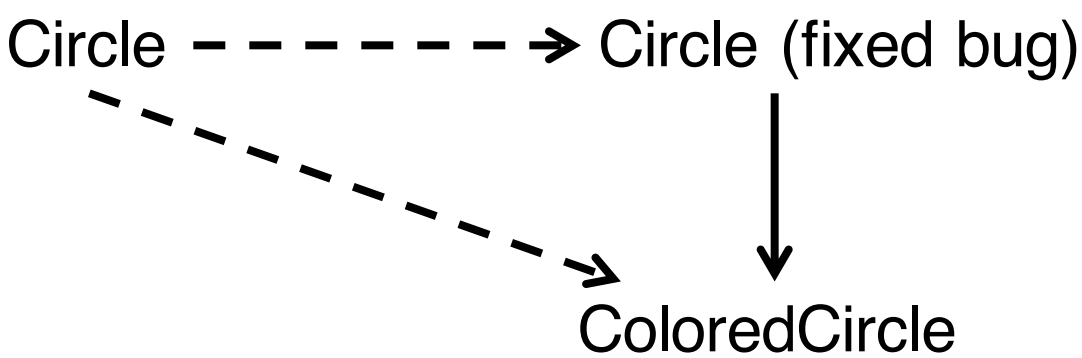
Option 1: copy-and-paste



Inheritance

```
class ColoredCircle extends Circle {
  private Color color;
  pubic ColoredCircle(int r, Color c)
        {super(r);color=c;}
  Color getColor {return color};
}
```

Constructor of superclass



Inheritance

```
class ColoredCircle extends Circle {
  private Color color;
  pubic ColoredCircle(double r, Color c)
        {super(r);color=c;}
  Color getColor {return color};
}
```

Inheritance introduces subtyping:

`ColoredCircle` inherits all fields & methods

`Circle` is the supertype of `ColoredCircle`

`ColoredCircle` is the subtype of `Circle`

Overriding

```
class DoubleCircle extends Circle {
  pubic DoubleCircle(double r){super(r);}
  public int edges {return 2};
}
```

Subclass may redefine methods in super class