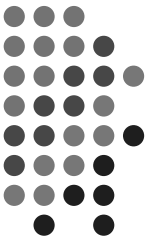
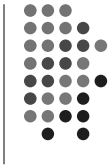
OOP
A Deeper Look

4A

Inheritance, Interfaces, and Abstract Classes

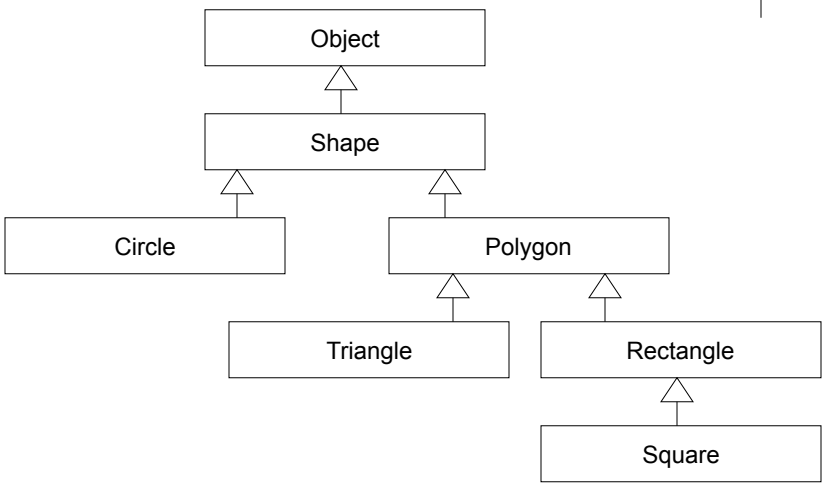




Inheritance

- All classes inherit from the `Object` class in Java.
- All classes are arranged in a hierarchy (a tree, more about this later in the semester) with `Object` at the top of the hierarchy.
- A class that inherits from another is called a subclass.
- A class that provides attributes and methods for inheritance by subclasses is called a superclass.

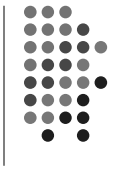
Superclasses & Subclasses



Inheritance keywords

- A class that `extends` another class is a subclass that inherits all fields and methods of the superclass.
- The subclass has direct access to all fields that are `public` and `protected`.
- The subclass can override the definitions of inherited methods with new implementations (using the same signature) and can access overridden methods using the `super` keyword.

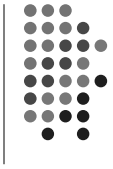
Example: A superclass



```
public class BankAccount
{
    private double balance;

    public BankAccount() {...}
    public BankAccount(double initBalance) {...}
    public void deposit(double amount) {...}
    public void withdraw(double amount) {...}
    public double getBalance() {...}
}
```

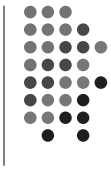
automatically inherits from `Object` if no other superclass is specified

Example: A subclass



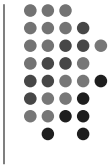
```
public class SavingsAccount extends BankAccount
{
    private double interestRate;
```

Constructors



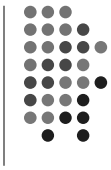
```
public SavingsAccount(double initRate,
                      double initBalance) {
   super(initBalance);
   interestRate = initRate;
}

public SavingsAccount(double initRate) {
   super();
   interestRate = initRate;
}
```


New method



```
public void addInterest() {
   deposit(getBalance()*interestRate);
}
```

Another way: `protected`



In `BankAccount`:

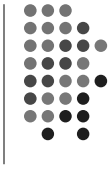
```
protected double balance;
```

In `SavingsAccount`:

```
public void addInterest() {
    deposit(balance*interestRate);
}
```

A field (or method) with protected visibility can be accessed in either the class that defines it, its subclasses or any class in the same package.

Overriding



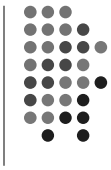
- A savings account subtracts a fee of $10 for withdrawals over $1000.
- We need to override the `withdraw` method and provide a new implementation that is appropriate.

```
public void withdraw(double amount) {
 if (amount > 1000.0)
     super.withdraw(amount + 10.0);
 else
     super.withdraw(amount);
}
```

same signature

same return type

Overriding Object methods

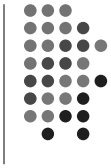


Classes inherit methods from the `Object` class. These methods typically do not work properly for our specific subclasses so we must override them.

```
public boolean equals(Object obj) {
      SavingsAccount other = (SavingsAccount)obj;
      return
           this.interestRate == other.interestRate
           && this.balance == other.balance;
}
```

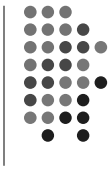
(assuming `balance` is `protected` in the `BankAccount` class)

Overriding Object methods



- Methods you should override:
  - `public boolean equals(Object obj)`
    Compares this object with the specified object by comparing the references only.
  - `public String toString()`
    Returns the class name + "@" + the hexadecimal representation of the object's hashcode.
  - `public int hashCode()`
    Calculates the hashcode of this object based on its reference only.

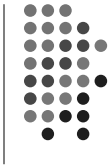
Polymorphism



```
BankAccount acct;
acct = new BankAccount(15111.0);
acct.withdraw(2000.0);
System.out.println(acct.getBalance());
acct = new SavingsAccount(15111.0);
acct.withdraw(2000.0);
System.out.println(acct.getBalance());
```

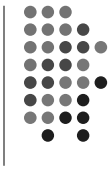
The same statement calls two different methods.

The `Stack<E>` class



- The **`java.util`** package includes a **`Stack<E>`** class.
- **`Stack<E>`** includes these **`public`** methods:
  - **`boolean empty()`**
  - **`E peek()`**
  - **`E pop()`**
  - **`E push(E item)`**
  - **`int search(Object obj)`**
- **`Stack<E>`** extends **`Vector<E>`**.
  - Why is this very poor design decision?

Interfaces



- A Java interface (not a GUI) is a means for defining specifications for behaviors that are common across classes that are not directly related by inheritance.

```
public interface Comparable<T> {
    int compareTo(T obj);
}
```

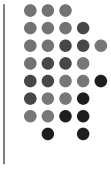
no fields

abstract method(s)

An interface cannot be instantiated directly: **NO**

```
Comparable<String> s = new Comparable<String>();
```

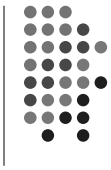

Using Interfaces



Any class that provides the behavior(s) specified in an interface must `implement` that interface and provide implementations for the methods specified.

```
public class BankAccount
         implements Comparable<BankAccount> {
     ...
     public int compareTo(BankAccount other) {
         ...  // provide implementation
     }
}
```

The `Queue<E>` interface

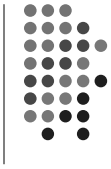


- The `java.util` package includes a `Queue<E>` interface.
- `Queue<E>` includes specifications for these methods:
  - `boolean offer(E item)` — enqueue
  - `E element()` * — peek
  - `E peek()` ** — peek
  - `E remove()` * — dequeue
  - `E poll()` ** — dequeue

  * Throws an exception if queue is empty.

  ** Returns null if the queue is empty.

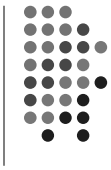
Using `Queue<E>`



- `LinkedList<E>` implements `Queue<E>`.
- Example: A queue of customers

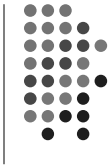
```
Queue<Customer> customerQ =
     new LinkedList<Customer>();
Customer c = new Customer("Andrew");
customerQ.offer(c);          // OK
customerQ.add(c);            // OK
customerQ.addFirst(c);       // BAD (Why?)
```

Abstract Classes



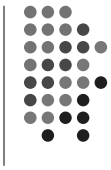
- An abstract class can have abstract methods like interfaces and also fields.
- Abstract classes can also have concrete (implemented) methods.
- An abstract class cannot be instantiated directly.
- Subclasses of an abstract class must provide implementations of the abstract methods specified in the abstract class definition.
- Abstract classes can have constructors that initialize fields defined in the abstract class.

Abstract Class Example



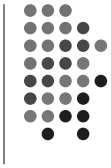
```
public abstract class Vehicle {
    private int speed;
    private String manufacturer;
    ...
    public int getSpeed { return speed; }
    public String getManufacturer
           { return manufacturer; }
    public abstract double computeTax();
    ...
}
```

Abstract Class Example (cont'd)



```
public class Truck extends Vehicle {
    private int numWheels;
    ...
    public double computeTax() {
       return 10.0 * numWheels;
    }
    ...
}
```


Correct Usage



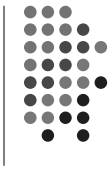
Interfaces:

```
String s = new String("Pittsburgh");
Comparable<String> s = new String("Erie");
```

Abstract classes:

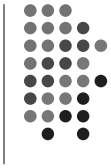
```
Truck myTruck = new Truck("Good Humor");
Vehicle myRide = new Truck("Ryder");
```

Comparison



	Actual Class	Abstract Class	Interface
Instances can be created	Y	N	N
Can define fields and methods	Y	Y	N
Can define constants	Y	Y	Y
Number of these a class can extend	0 or 1	0 or 1	0
Number of these a class can implement	0	0	any number
Can extend another class	Y	Y	N
Can declare abstract methods	N	Y	Y
Can declare variables of this type	Y	Y	Y

Casting

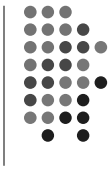


- In the `equals` method, we used the following casting:

  ```
  SavingsAccount other = (SavingsAccount)obj;
  ```

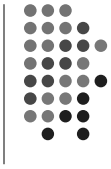
- What if `obj` isn't a `SavingsAccount`?
  - A `ClassCastException` is generated during runtime.
- We can use the `instanceof` operator to check the object's actual type during runtime.

`instanceof` example



```
public boolean equals(Object obj) {
    if (obj instanceof SavingsAccount) {
       SavingsAccount other = (SavingsAccount)obj;
       return
         this.interestRate == other.interestRate
         && this.balance == other.balance;
    }
    return false;
}
```

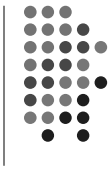

Copying data



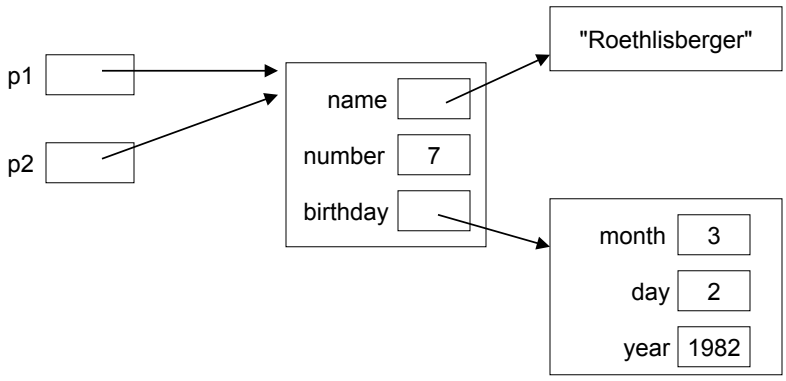
```
public class Player {
    String name;
    int number;
    CalendarDate birthday;
}
```

```
Person p1 = new Person("Roethlisberger", 7,
             new CalendarDate(3, 2, 1982);
```

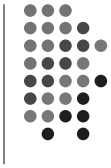
Copying data



```
Person p2 = p1;
```



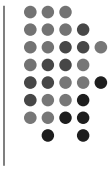
Cloning: Shallow Copy



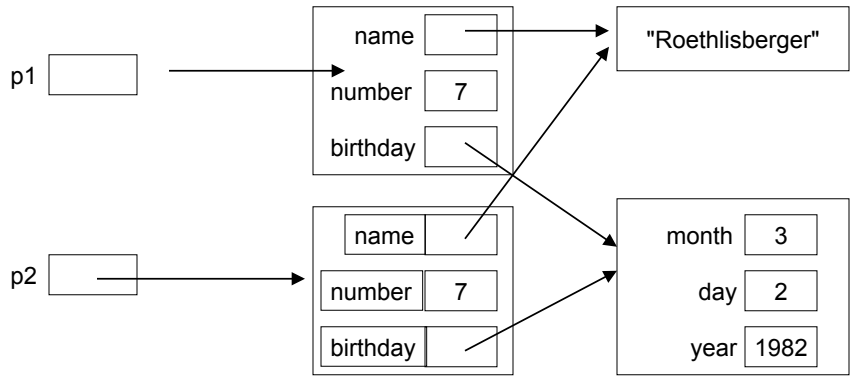
In the `Person` class:

```
public Object clone() {
    Object newObj
            = new Person(name, number, birthday);
    return newObj;
}
```

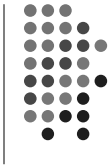
Cloning: Shallow Copy



```
Person p2 = (Person)p1.clone();
```



Cloning: Deep Copy



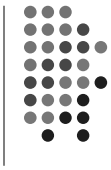
In the `Person` class:

`Object` has a `clone` method that performs the shallow copy.

```
public Object clone() {
    try {
       Person newPerson = (Person)super.clone();
       newPerson.birthday =
            (CalendarDate)birthday.clone();
       return newPerson;
    } catch (CloneNotSupportedException e) {
       throw new InternalError();
    }
}
```

`CalendarDate` must also have a `clone` method.

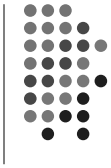
Cloning: Deep Copy



In the `CalendarDate` class:

```
public Object clone() {
    try {
       CalendarDate newDate =
            (CalendarDate)super.clone();
       return newDate;
    } catch (CloneNotSupportedException e) {
       throw new InternalError();
    }
}
```


Cloning: Deep Copy

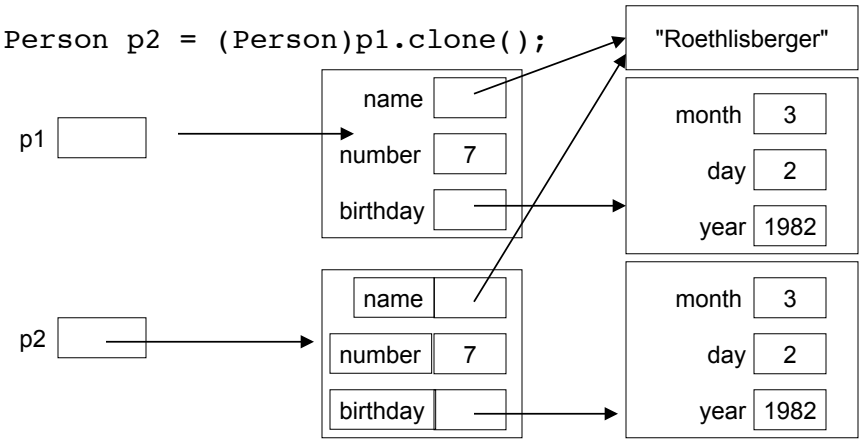


The method `Object.clone` will generate a `CloneNotSupportedException` if it is called in a class that does not `implement` the `Cloneable` interface.

Therefore:

```
public class Person implements Cloneable {
      ...
}
public class CalendarDate implements Cloneable {
      ...
}
```

Cloning: Deep Copy



Cloning: Deep Copy

Strings are immutable