Mostly Review
(Adapted to Java 5)
Java Types

- Every variable has a type (strong typing)
- There are two types of variables
  - Eight primitive types
    - byte, char, short, int, long, float, double, boolean
    - They are not objects
  - Reference types
    - interface type (Translatable, Comparable)
    - object type (Point, BankAccount)
Wrapper classes (pre Java 5)

- We need to use primitive types as objects
- This is done using wrapper classes
  - Byte, Character, Short, Integer
  - Long, Float, Double, Boolean

```java
public Integer(int v);
public int intValue();
```

```java
Integer iObj = new Integer(34);
int i = iObj.intValue();
```
Wrapper classes (Java 5)

Java 5 has auto boxing and unboxing

- Boxing
  - Auto create wrapper object

- Unboxing
  - Auto convert wrapper object to primitive value

```java
public Integer(int v);
public int intValue();
```

```java
Integer iObj = 34;
int i = iObj;
```
Box and Arrow Notation

**Primitive types**
- area holds a double value
  - area: 3.14159

**Object types**
- greeting holds a reference to a String object
  - greeting
  - String
    - Hello
a = b for primitive types

- Assign the value of b to the value of a
- After a = b both a and b have the value of b

Before a = b

<table>
<thead>
<tr>
<th>a</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>19</td>
</tr>
</tbody>
</table>

After a = b

<table>
<thead>
<tr>
<th>a</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>19</td>
</tr>
</tbody>
</table>
a = b for reference types

- After assignment a and b are both refs to the same object (one referenced by b)
- a and b are said to be aliases

Before a = b

After a = b
== for objects

- The == operator compares references for equality not objects.
- \( a == b \) is true only if \( a, b \) are aliases for the same object.

![Diagram showing object references]

\( a == b \) is true here.
Object equality

Suppose "object a" and "object b" have the same type and values for all data fields.

Let a and b be references to these objects

a == b will be false so we need an equals method such that a.equals(b) will be true. This method tests for object equality.
Overriding equals

- `equals` is defined in the `Object` class so every class has an `equals` method with prototype
  
  - `public boolean equals(Object obj)`

- Version in the `Object` class is designed so that `a.equals(b)` is the same as `a == b`

- This is normally not what we want so it is necessary to override `equals` using our definition of object equality
Define two Point objects to be the same if they have the same x and y coordinates

```java
package pointexamples;
public class Point
{
    private double x, y;
    ...
    public boolean equals(Object obj) // overriden
    {
        if (obj instanceof Point)
        {
            Point p = (Point) obj;
            return (p.getX() == x) && (p.getY() == y);
        }
        return false;
    }
    ...
}
```
Define two BankAccount objects to be the same if they have the same account number and name

```java
package bankexamples.accounts;
public class BankAccount implements Cloneable, Serializable {
    ...
    public boolean equals(Object obj) // overridden
    {
        if (obj instanceof BankAccount)
        {
            BankAccount b = (BankAccount) obj;
            return (b.getNumber() == getNumber()) &&
                    (b.getName().equals(getName())) ;
        }
        return false;
    }
}
```
Define two JointBankAccount objects to be the same if they have the same account number, same name, and same joint name.

```java
package bankaccounts.accounts;
public class JointBankAccount extends BankAccount {
    ...
    public boolean equals(Object obj) {
        if (obj instanceof JointBankAccount) {
            JointBankAccount b = (JointBankAccount) obj;
            return super.equals(b) && getJointName().equals(b.getJointName());
        }
        return false;
    }
}
```
Using the equals method

```java
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);

if (p1 != p2)
{
    System.out.println("Not equal");
}

if (p1.equals(p2))
{
    System.out.println("Objects are equal");
}
```
Overriding clone

- The clone method has prototype
  - `protected Object clone()`
- It exists in the `Object` class
- Default behaviour is to clone the instance data fields of an object (shallow clone)
- We can override `clone` and make it a public method to copy (clone) an object
- The class must implement the `Cloneable` interface
- In Java 5 a covariant return type can be used
Point class should implement Cloneable if it wants to override the Clone method in Object class

```java
package pointexamples;
public class Point implements Cloneable {
    private double x, y;
    ...
    public Point clone() // covariant return type
    {
        Point copy;
        try {
            try {
                copy = (Point) super.clone();
            } catch (CloneNotSupportedException e) {
                throw new RuntimeException(
                        "Cannot clone Point object");
            }
        }
        return copy;
    }
}
```
BankAccount clone method

BankAccount class should implement Cloneable if it wants to override the Clone method in Object class

package bankexamples.accounts;
public class BankAccount implements Cloneable {
    ...
    public BankAccount clone()
    {
        BankAccount copy;
        try
        {
            copy = (BankAccount) super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            throw new RuntimeException(
                "Cannot clone BankAccount object");
        }
        return copy;
    }
}
JointBankAccount clone

Since JointBankAccount is a subclass of BankAccount the clone method is simpler

```java
package bankexamples.accounts;
public class JointBankAccount extends BankAccount {
    ...
    public JointBankAccount clone() {
        return (JointBankAccount) super.clone();
    }
    ...
}
```
Using the clone method

- Two references to the same object
  ```java
  Point p1 = new Point(1,2);
  Point p2 = p1;
  ```
  Here `p1 == p2` is true

- Two references to different equal objects
  ```java
  Point p1 = new Point(1,2);
  Point p2 = p1.clone();
  ```
  Here `p1 == p2` is false
Alternatives to clone

- Use a constructor to make a copy
  
  ```java
  Point p1 = new Point(1,2);
  Point p2 = new Point(
      p1.getX(), p2.getY());
  ```

- Use a factory method
  
  ```java
  Point p2 = Point.getCopy(p1);
  ```
Class/Interface Relations (1)

- Class/Interface inheritance
  - "is a"

- Interface implementation
  - "is a", "implements a"

- Aggregation (similar to Composition)
  - "has a", "is a part of", "contains a"

- Dependency (a kind of association)
  - "uses"
Class/Interface Relations (2)

- Class inheritance in Java
  - ClassA extends ClassB

- Interface inheritance in Java
  - InterfaceA extends InterfaceB

- Interface implementation in Java
  - ClassA implements InterfaceA
  - Like inheritance this is also an "is a" relation
Class/Interface Relations (3)

- Aggregation of objects
  - "has a" relation
  - ClassA has an instance data field reference to ClassB
    - ClassA "has a" ClassB
  - "is a part of " inverse relation
    - ClassB " is a part of " ClassA

- Example
  - A Circle object "has a" Point object (center of the circle) or Point " is a part of " Circle
Class/Interface Relations (4)

- **Dependency** (a kind of association)
  - ClassA depends on ClassB
  - ClassA uses ClassB
- A method or constructor in Class A requires an object from Class B as a parameter
- A method in Class A constructs a local variable referring to an object from ClassB
Class design/implementation

- **Writing a class has two distinct parts**
  - Writing a design or specification
  - Writing an implementation

- **Specification (what) (e.g. lists)**
  - What does the class do? not how
  - Javadoc or an interface provides the specification

- **Implementation (how)**
  - The internal details and the code
**Example**

- Specification of the withdraw method in the `BankAccount` class

```java
/**
   * Javadoc specification goes here
   */
public void withdraw(double amount)
```

- Implementation of `withdraw` method.

```java
public void withdraw(double amount) {
    // check that amount is legal
    balance = balance - amount;
}
```
Data encapsulation (1)

- In OOP the data is spread out over the instance data fields of one or more objects.
- Each object is only responsible for its own data.
- This data is normally private so that it cannot be directly changed from outside the class.
- This is called **data encapsulation**.
Data encapsulation (2)

- Classes can be **mutable** or **immutable**
- For mutable classes the constructors and methods that change the values of one or more instance data fields can be written to check that objects are not in an inconsistent state.
- If possible make classes immutable
- For mutable classes involving composition unwanted side effects can occur.
Side effects

- Changing the state of an object indirectly from outside the class
- Often this is desirable (BankAccount)
- Unwanted side-effects can be created by returning references to mutable instance data fields from methods or using parameters
- Such a reference can then be used outside the class to change the state of an object.
Point and MPoint classes

**Point** (immutable class)
- represents points in the plane. It has `getX` and `getY` methods but no `setX` and `setY` methods.

**MPoint** (mutable class)
- Like **Point** but has `setX` and `setY` methods

```java
public void setX(double x) {
    this.x = x;
}
public void setY(double y) {
    this.y = y;
}
```
public class Circle
{
    private Point center;
    private double radius;

    public Circle(Point p, double r)
    {
        center = p; radius = r;
    }

    public Circle(double x, double y, double r)
    {
        center = new Point(x, y); radius = r;
    }

    public Point getCenter()
    {
        return center;
    }

    // other methods
}

This class is immutable since Point is immutable
public class MCircle {
    private MPoint center;
    private double radius;

    public MCircle(MPoint p, double r) {
        center = p; radius = r; }

    public MCircle(double x, double y, double r) {
        center = new MPoint(x, y); radius = r; }

    public MPoint getCenter() {
        return center; }

    // other methods
}
Side effects in MCircle

```java
public MCircle(MPoint p, double r) {
    center = p;
    radius = r;
}
```
center can be changed through external ref p

```java
public MPoint getCenter() {
    return center;
}
```
center can be changed through ref returned
Example of side-effect (1)

```java
MPoint p = new MPoint(3,4);
MCircle c = new MCircle(p, 5);
System.out.println(c);
```

The result above is a circle with center (3,4) and radius 5

```java
p.setX(999);
System.out.println(c);
```

The result above is a circle with center (999,4) and radius 5

We have changed the state of the circle by manipulating the point p which is outside the MCircle class. This is probably an unwanted side-effect.
Example of side-effect (2)

```java
MCircle c = new MCircle(3, 4, 5);
System.out.println(c);

Here we have used the other constructor so there are no side effects here. The result above is a circle with center (3,4) and radius 5

MPoint p = c.getCenter();
p.setX(999);
System.out.println(c);

The result above is a circle with center (999,4) and radius 5 since the returned reference from getCenter is a reference to the data field in MCircle
```
Example of side-effect (3)

The problem occurs with the statements
- `center = p;  // in constructor`
- `return center;  // in getCenter method`

In either case we have two references to the same MPoint:
- One reference is inside the class (center)
- The other reference is outside the class (p)
Example of side-effect (4)

There are two references to one MPoint object.
No side-effect in original classes

Point p = new Point(3,4);
Circle c = new Circle(p, 5);
System.out.println(c);

Point p = c.getCenter();
p = new Point(999,4);
System.out.println(c);

Here both println statements show that the circle center remains at (3,4) since we do not have set methods in Point class.

The references p and center are to different objects so the Circle class has its own Point object.
Fixing the side-effect (1)

- One way is to include a copy constructor in `MPoint` class

  ```java
  public MPoint(MPoint p) {
      x = p.x;
      y = p.y;
  }
  ```

- Or use `new MPoint(p.getX(), p.getY())` to make a new point or use `clone` if the class has implemented it.
Fixing the side-effect (1)

- Revise constructor that takes an `MPoint` object as argument (or use `p.getX, p.getY`)
  
  ```java
  public MCircle(MPoint p, double r) {
    center = new MPoint(p);
    radius = r;
  }
  ```

- Revise `getCenter` method
  
  ```java
  public MPoint getCenter() {
    return new MPoint(center);
  }
  ```

  Now center references its own private copy of the `MPoint` object.
  Now we return a copy of the private `MPoint` object.
Side-effects are gone

```java
MPoint p = new MPoint(3,4);
MCircle c = new MCircle(p, 5);
System.out.println(c);

MPoint p = c.getCenter();
p.setX(999);
System.out.println(c);
```

The `setX` statement now has no effect on the point inside the `MPoint` object since we now have two references to two different objects instead of one object.

`MPoint` now has its own copy of its center point.
Now \( p \) and \( \text{center} \) are references to different objects.
Most examples of useful side effects involve method reference arguments:

Consider a bank account transfer method

```java
public void transfer(BankAccount from, BankAccount to, double amount)
{
    from.withdraw(amount);
    to.deposit(amount);
}
```

Here we are modifying two account objects using references to them.
Comparable interface (1)

For some classes the objects have some kind of default or natural order.

Given two objects $a$, $b$ then there are three possibilities:

- $a$ precedes $b$
- $a$ equals $b$
- $a$ follows $b$

Example natural order of positive integers is

- $1 < 2 < 3 < 4 < 5 < ...$
Comparable interface (2a)

This interface is in package `java.lang` and has only one method:

```java
public interface Comparable
{
    public int compareTo(Object obj);
}
```

Then `a.compareTo(b)` returns:

- negative number if `a` precedes `b`
- zero if `a` is equal to `b`
- positive integer if `a` follows `b`
Comparable interface (2b)

This interface is in package `java.lang` and has only one method

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

Then `a.compareTo(b)` returns

- negative number if `a` precedes `b`
- zero if `a` is equal to `b`
- positive integer if `a` follows `b`
Comparable interface (3)

- The `String` class implements this interface.
- We have already used boolean expressions for strings such as:
  - `s1.compareTo(s2) < 0`
  - `s1.compareTo(s2) == 0`
  - `s1.compareTo(s2) > 0`
- If the first expression is true it means that `s1` precedes `s2` in the lexicographical ordering of strings.
- `equals` can be used in the `==` case.
To make BankAccount implement the Comparable interface include the following method which assumes that the standard ordering is increasing order by account number

```java
package bankexamples.accounts;
public BankAccount implements ... , Comparable {
    public int compareTo(Object obj) {
        if (!(obj instanceof BankAccount)) {
            throw new IllegalArgumentException("Cannot compare objects");
        }
        BankAccount b = (BankAccount) obj;
        return getNumber() - b.getNumber();
    }
}
```
Types

- In Java we have object and interface types.
- An inheritance hierarchy is a type.
- A set of classes that implement an interface is a type.
- Interface types are more general than types defined by inheritance.
- The "is a" relation is used to describe types.
  - A JointBankAccount "is a" BankAccount.
Inheritance type casting (1)

- Type casting up an inheritance hierarchy is always possible without an explicit typecast
  - `JointBankAccount jb = new JointBankAccount(...);`
  - `BankAccount b = jb;`

- This is object amnesia
  - `jb` forgets that it is a `JointBankAccount` and doesn't remember that it has a `getJointOwner` method
Inheritance type casting (2)

- Following statements are legal
  - int number = jb.getNumber();
  - int number = b.getNumber();

- Following statement is illegal
  - String jointName = b.getJointName();

- Following statement is legal (type cast reminds b that its a JointBankAccount object)
  - String jointName = ((JointBankAccount) b).getJointName();
Interface type casting

- If a class implements an interface then it can be cast to the interface type without explicit type casting

Example: If `BankAccount` implements the `Comparable` interface then every `BankAccount` object "is a" `Comparable` object:

  - `BankAccount b = new BankAccount(...) ;`
  - `Comparable c = b ;`
Polymorphism

Polymorphic type
- set of classes in an inheritance hierarchy
- set of classes that implement an interface

Polymorphic method
- method that has the same prototype in each class in the type.
- can have a different implementation for each class in the type.
  - this is done by overriding the method
Explicit type casting (1)

- Casting down a type hierarchy

Example:

- `BankAccount b1 = new BankAccount(...);`
- `BankAccount b2;`
- `Object obj = b1; // b1 thinks its an Object`
- `b2 = obj; // illegal: obj is not a BankAccount`
- `b2 = (BankAccount) obj; // legal`

But note that it is necessary that obj really be a BankAccount object else a class cast exception occurs at run time
Explicit type casting (2)

Example: suppose `Person` is an interface and `Student` is a class implementing this interface. Then we can cast

```java
Student s = (Student)p; // p is a Person
```

This works at run time even though a `Person` is not a `Student` (a `Student` is a `Person`) if `Student` is the only kind of `Person` object
1-D arrays: random access

- Random access data structure
  - Each element can be accessed in $O(1)$ time
- Each array element occupies a block of consecutive bytes in memory.
- If an element takes $size$ bytes of memory then the location of element $a[k]$ is
  - $a[0] + k \times size$

so arrays are random access structures
Array addressing (int array)

This explains why random access is efficient for arrays.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>34</td>
</tr>
<tr>
<td>a[1] = a[0] + 4*1</td>
<td>45</td>
</tr>
<tr>
<td>a[2] = a[0] + 4*2</td>
<td>67</td>
</tr>
<tr>
<td>a[3] = a[0] + 4*3</td>
<td>33</td>
</tr>
<tr>
<td>a[4] = a[0] + 4*4</td>
<td>9</td>
</tr>
</tbody>
</table>

General case: \( a[k] = a[0] + 4*k \)
Pictorial representation (1)

create reference to array

int[] score;

Create the array

score = new int[5];

assign values to the array elements

score[0] = 1000;
...

BGA
Arrays of object references

The array is an array of object references
The objects are stored on the heap
Create the array of references

null
null
null
b[0]
b[1]
b[2]

assign object references to array elements

null
null
null
b[0]
b[1]
b[2]

create reference to array
Adapter class

- The methods of an adapter class are implemented using composition or aggregation and the methods of another class (delegation).
- This is called the adapter design pattern.
- The idea is to adapt a complex class to be easier to use for some specific purpose.
- Adapter classes are used a lot.
The Calendar class is a complex class for manipulating dates and times.

```java
public class CalendarMonth {
    private Calendar calendar;

    public CalendarMonth() {...}
    public CalendarMonth(int year, int month){...}
    public int dayOfWeek() {...}
    public int daysInMonth() {...}
    public String monthName() {...}
    ...
}
```

We did this example in first year.
Collection Class

- A collection class contains objects of some class or type
- The collection class groups these objects together
- Normally the objects in a collection class all have something in common.
- Example: Collection Class of BankAccount objects or Point objects
Arrays as collection classes

The array data structure provides the simplest way to group together some related objects.

Example: Array for 100 bank accounts

```
BankAccount[] bank = new BankAccount[100];
```

Operations

```
b[k] = new BankAccount(...); // add an account
BankAccount b = b[k] // get an account
```

We also used ArrayList briefly in COSC 1047
Bank Example

Make a very simple **Bank** collection class:
- hides the fact that an array is being used
- uses an array as a private data field to store the collection of objects.

Operations (minimal)
- create a **Bank** object
- **add** method to add a new account at end
- **get**, **set** methods to access the accounts (this is an indexed collection).
A very simple collection class for some BankAccount objects. We can construct an empty bank, add accounts to it, get an account, and set a new account (replace)

```java
package bankexamples.simple;
import bankexamples.accounts.BankAccount;
public class Bank implements Cloneable {
    // Fixed size array implementation so we must always have
    // 0 <= size <= maxSize
    // bank[0] to bank[size-1] is the part of the array in use.
    // A new account is added at bank[size] if there is room
    private BankAccount[] bank;
    private int size; // number of accounts
    private int maxSize; // max number
}
```
Construct a default bank for a maximum of 10 accounts

```java
public Bank()
{
    this(10);
}
```

Construct a bank for a maximum of `maxSize` accounts

```java
public Bank(int maxSize)
{
    bank = new BankAccount[maxSize];
    size = 0;
    this.maxSize = maxSize;
}
```
Return the size of the bank. This is the number of accounts currently in the bank. Note that we need size() in order to indicate how much of the array is being used.

```java
public int size()
{
    return size;
}
```
Add another account to the bank after the last one

```java
public void add(BankAccount b) {
    if (size == maxSize) {
        throw new IllegalArgumentException("bank is full");
    }
    // size marks next position
    bank[size] = b;
    size++;
}
```
public BankAccount get(int index) {
    if (index < 0 || index >= size()) {
        throw new IllegalArgumentException("invalid index for get");
    }
    return bank[index];
}
Replace object at given index with a new object and return the original object at this position

```java
public BankAccount set(int index, BankAccount b)
{
    if (index >= size() || index < 0)
    {
        throw new IllegalArgumentException(
            "invalid index for set");
    }
    BankAccount original = bank[index];
    bank[index] = b;
    return original;
}
```
Clone a Bank object
Note: Arrays have a built-in clone method

```java
public Bank clone()
{
    Bank copy;
    try
    {
        copy = (Bank) super.clone();
    }
    catch (CloneNotSupportedException e)
    {
        throw new RuntimeException(
            "Cannot clone Bank object");
    }
    copy.bank = bank.clone();
    return copy;
}
```
The `toString` method returns list of bank accounts with new lines between each account.

```java
public String toString()
{
    if (size() == 0) return "Empty list";
    StringBuilder s = new StringBuilder();
    for (int k = 0; k < size(); k++)
    {
        s.append(bank[k]);
        s.append("\n");
    }
    return s.toString();
}
```
Create a small bank and display the accounts

```java
Bank bank = new Bank(5);
bank.add(new BankAccount(123, "Fred", 100.0));
bank.add(new BankAccount(124, "Mary", 150.0));
bank.acc(new BankAccount(125, "Gord", 200.0));
System.out.println(bank);
```
Bank testing (2)

Use indexed get method to display that bank accounts
A for loop can be used to traverse the list of accounts

for (int k = 0; k < bank.size(); k++)
{
    System.out.println(bank.get(k));
}
Bank testing (3)

Traverse the list of accounts and add $100 to each account. This is a desirable side effect.

for (int k = 0; k < bank.size(); k++)
{
    BankAccount b = bank.get(k);
    b.deposit(100);
}

Note that the bank holds only references to the accounts so it is possible to change the account at a given position using the deposit and withdraw methods.
Use set to replace each account with one having same name and balance but account numbers 1, 2, 3, 4, ...

```java
for (int k = 0; k < bank.size(); k++) {
    BankAccount b = bank.get(k);
    BankAccount a = new BankAccount(k+1,
    b.getName(), b.getBalance();
    bank.set(k, a);
}
```
Other things to test

- bank is empty
  - test `get` and `set` in this case
- bank is not full and not empty
  - test `get` and `set` with an out of range index to see if exception is thrown.
- bank is full
  - test `add` method to see if exception is thrown.

Bank can store null objects
Bank testing (6)

- Note that we have used the `BankAccount` class in the `Bank` class.
- Because `JointBankAccount` is a subclass of `BankAccount` we can also put joint accounts into the bank.
- This should be tested by creating a bank that has both types of accounts in it.
Cloning an integer array

Original Array

Clone of Original Array

The following assignment only clones the array reference

\[ b = a; \ // \text{two references to the same object} \]

To clone the array as shown in the picture it is necessary to use

\[ b = a.clone(); \ // \text{No type cast in Java 5} \]
Cloning BankAccount arrays

Note that for an array of objects the array is cloned but the objects are not cloned. This is the default behaviour and is called a shallow clone. We say that the objects are shared.

```
BankAccount[] b = a.clone();
```
To clone bank objects from the class Bank we can insert the following clone method in class Bank

```java
public Bank clone()
{
    Bank copy;
    try
    {
        copy = (Bank) super.clone();
    } catch (CloneNotSupportedException e) {
    
        throw new RuntimeException(
            "Cannot clone Bank object");
    }
    copy.bank = bank.clone();
    return copy;
}
```
Cloning the bank (2)

Diagram showing the cloning of a bank. The original bank has a size of 3 and a maxSize of 5. The cloned bank, bankClone, has the same structure and contains the same accounts as the original bank, with Fred, Mary, and Gord having balances of 100.0, 150.0, and 200.0 respectively.
Testing the bank clone (1)

See tester class bankexamples.simple.BankTester

Construct a bank with 3 accounts and clone it

```java
Bank bank = new Bank(5);
bank.add(new BankAccount(123, "Fred", 100.0));
bank.add(new BankAccount(124, "Mary", 150.0));
bank.add(new BankAccount(124, "Gord", 200.0));

Bank bankClone = bank.clone();

System.out.println(bank);
System.out.println(bankClone);
```

The results displayed are the same for bank and bankClone
Testing the bank clone (2)

Add $1000 to Fred's account

```java
BankAccount b = bank.get(0);
b.deposit(1000);
System.out.println(bank);
System.out.println(bankClone());
```

Again the same results are displayed for bank and bankClone since they share the same BankAccount objects
Testing the bank clone (3)

Replace the account with index 0 in bank by a new account.

```java
bank.set(0, new BankAccount(123, "Bill", 0));
System.out.println(bank);
System.out.println(bankClone());
```

Now the results are different since we changed one of the references in bank.

This does not affect the references in bankClone
Modifying the bank (1-4)

bank

bank
size 3
maxSize 5

bankClone

bank
size 3
maxSize 5

123 Fred 100.0
124 Mary 150.0
125 Gord 200.0
Modifying the bank (2-4)

Bank:
- Name: Bill
- Size: 0.0

Bank Clone:
- Name: Fred
- Size: 1100

Bank:
- Name: Mary
- Size: 150.0

Bank:
- Name: Gord
- Size: 200.0
Modifying the bank (3-4)
Modifying the bank (4-4)

**Diagram:**

- **Bank:**
  - **size:** 3
  - **maxSize:** 5

- **Bank Clone:**
  - **size:** 3
  - **maxSize:** 5

- **Transactions:**
  - 123 Bill 0.0
  - 123 Fred 1100
  - 124 Mary 150.0
  - 125 Gord 200.0

**Notes:**

- Fred
- Mary
- Gord

**Date:** 8/23/2007
Deep clone

- The Bank clone is called a shallow clone
- It clones the array of references
- It does not make copies of the accounts themselves which would give a deep clone
Deep cloning the bank

bank

bank
size
maxSize

bankClone

bank
size
maxSize

123 Fred 100.0
123 Fred 100.0
124 Mary 150.0
124 Mary 150.0
125 Gord 200.0
125 Gord 200.0
The deep clone version of the clone method needs a for loop to clone the accounts using the BankAccount clone method.

```java
public Bank clone()
{
    Bank copy;
    try
    {
        copy = (Bank) super.clone();
    } catch (CloneNotSupportedException e)
    {
        throw new RuntimeException(
            "Cannot clone Bank object");
    }
    copy.bank = bank.clone();
    for (int k = 0; k < size(); k++)
    {
        copy.bank[k] = bank[k].clone();
    }
    return copy;
}
```
Unified modelling language

Part of UML includes a notation for expressing relationships among classes
- inheritance "is a"
- implements "is a"
- composition "is a part of", "has a"
- association "uses"

BlueJ uses a simple form of UML diagrams for classes
UML Diagram notation

Class Diagram

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Class Name</th>
<th>Interface Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance variables</td>
<td>&lt;&lt;abstract&gt;&gt;</td>
<td>（）methods</td>
</tr>
<tr>
<td>methods</td>
<td>Instance variables</td>
<td>methods</td>
</tr>
</tbody>
</table>

Inheritance ("is a")

Implements ("is a")

Composition ("is a part of")

Association ("uses")

8/23/2007  BGA
UML class diagram

BankAccount

| number   |
| name    |
| balance |
| + deposit |
| + withdraw |
| + getNumber |
| + getName |
| + setName |
| + getBalance |
| + compareTo |
| + equals |
| + clone |
| + toString |

JointBankAccount

| jointName |
| + getJointName |
| + setJointName |
| + getBalance |
| + equals |
| + clone |
| + toString |

Bank

| size |
| + add |
| + get |
| + set |
| + clone |
| + toString |

Comparable

| <<interface>> |
| + compareTo |

1..m "is a part of" "is a" "implements"
Javadoc rules

- /** ... */ denote javadoc comments
- put one in front of each public class, constructor and method
- @param documents a parameter
- @throws documents exceptions
- @return documents the return value
- @pre, @post custom data structures tags
- Many other tags
Class comment should describe the class and provide a description of how the class is used. Optionally it can include @author and @version tags.

```java
package ....;
import ....;

/**
 * Class comment goes here.
 * @author Fred Jones, September 11, 2005
 * @version 1.3
 */

public class MyClass
{
...
}
```
A method or constructor comment should begin with a sentence ending with a period. Additional comments can follow. Use @param, @throws, @return tags to describe parameters, exceptions, and return value.

/**
 * First sentence gives brief description.
 * Additional sentences follow if necessary.
 * @param paramName1 description.
 * @param paramName2 description.
 * @throws ExceptionClassName description.
 * @return description.
 * @pre description. (this is a custom tag)
 * @post description. (this is a custom tag)
 */
Defining your own exceptions

Later we will define our own Exceptions using inheritance:

```java
package bankexamples.exceptions;
public class MyException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public MyException(String message) {
        super(message);
    }
}
```
Java 5 scanner class (1)

- Can use it to get keyboard input
- Differs from our KeyboardReader class
- Does not follow the one-input per line input model.
- Can read several items spread out over several lines with one or more items per line
Java 5 scanner class (2)

- Scanner class is in package java.util
- Constructing a scanner for standard input:
  - Scanner input = new Scanner(System.in);
- Methods for reading items
  - String nextLine()
  - double nextDouble()
  - int nextInt()
- Many other examples.
If you want to follow the one input per line input model that we did with KeyboardReader it is necessary to follow each numeric input item with a `nextLine()` so that the new lines are eaten up.

```java
Scanner input = new Scanner(System.in);
System.out.println("Enter your age");
int age = input.nextInt();
nextLine(); // eat the newline

System.out.println("Enter your name");
String name = input.nextLine();

System.out.println("Enter your favourite food");
String food = input.nextLine();
```
Testing

- Choosing a good set of test cases is important
- Writing tester classes
- Writing GUI interfaces to test a collection class (buttons for methods and text fields to get input data)
- Automated unit testing using JUnit which is now a standard Eclipse plugin.
Use box like this for comment

Put code here