CS 1110
Prelim II: Review Session

(Fall'07) Question 1 (15 points). Write the body of the following function recursively.

```
/** = n, but with its digits reversed.
      Precondition: n >= 0.
      e.g. n = 135720, value is "027531".
      e.g. n = 12345, value is "54321".
      e.g. n = 7, value is "7".
      e.g. n = 0, value is "0".*/
public static String rev(int n) {



}
```

returns a String

Recursive Function 4 Principles

- 1. Write the precise specification


```
/** = n, but with its digits reversed.
      Precondition: n >= 0.
   e.g. n = 135720, value is "027531".
   e.g. n = 12345, value is "54321".
   e.g. n = 7, value is "7".
   e.g. n = 0, value is "0".*/
public static String rev(int n) {
   // base case:
   //{n has only one digit}


   // recursive case:
   // {n has at least two digits}

}
```


Recursive Function 4 Principles

- 1. Write the precise specification
- 2. Base Case


```
/** = n, but with its digits reversed.
      Precondition: n >= 0.
   e.g. n = 135720, value is "027531".
   e.g. n = 12345, value is "54321".
   e.g. n = 7, value is "7".
   e.g. n = 0, value is "0".*/
public static String rev(int n) {
   // base case:
   //{n has only one digit}
   if (n < 10)

   // recursive case:
   // {n has at least two digits}

}
```

Let's review some type issues

What is the type of?

- `42`
- `"" + 42;`
- `'a' + 'b'`
- `'b' + "anana"`
- `'b' + 'a' + "nana"`
- `'b' + ('a' + "nana")`
- `"" + 'b' + 'a' + "nana"`

7

```
/** = n, but with its digits reversed.
      Precondition: n >= 0.
   e.g. n = 135720, value is "027531".
   e.g. n = 12345, value is "54321".
   e.g. n = 7, value is "7".
   e.g. n = 0, value is "0".*/
public static String rev(int n) {
   if (n < 10)
      return "" + n;

   // recursive case:
   // {n has at least two digits}

}
```

base case: n has 1 digit

8

Recursive Function 4 Principles

- 1. Write the precise specification
- 2. Base Case
- 3. Progress
  - Recursive call, the argument is "smaller than" the parameter. Ensures base case will be reached (which terminates the recursion)
- 4. Recursive case

9

```
/** = n, but with its digits reversed.
      Precondition: n >= 0.
   e.g. n = 135720, value is "027531".
   e.g. n = 12345, value is "54321".
   e.g. n = 7, value is "7".
   e.g. n = 0, value is "0".*/
public static String rev(int n) {
   if (n < 10)
      return "" + n;

   // n has at least 2 digits
   return (n%10) + rev(n/10);

}
```

base case: n has 1 digit

recursive case:

10

```
/** = the reverse of s.*/
public static String rev(String s) {
   if (s.length() <= 1)
      return s;

   // { s has at least two chars }
   int k= s.length()-1;
   return s.charAt(k) +
          rev(s.substring(1,k)) +
          s.charAt(0);
```

base case

recursive case

Do this one using this idea:
To reverse a string that contains at least 2 chars, switch first and last ones and reverse the middle.

11

CS1110 Flix

12

```
public class Movie {
  private String title; // title of movie
  private int length; // length in minutes
  /** Constructor: document with title t
      and len minutes long */
  public Movie(String t, int len) {
    title= t; length= len;
  }
  /** = title of this Movie */
  public String getTitle()
    { return title; }
  /** = length of document, in minutes */
  public int getLength()
    { return length; }
  /** = the popularity:
      shorter means more popular */
  public int popularity()
    { return 240 - length; }
}
```

```
public class Documentary extends Movie {
  private String topic; // …

  /** Constructor: instance with title t,
      length n, and topic p */
  public Documentary(String t, int n,
                     String p) {
    super(t, n);
    topic= p;
  }

  /** = "Documentary" */
  public String DocumentaryType()
    { return "Documentary"; }

  /** = popularity of this instance */
  public int popularity()
    { return 200 - getLength(); }
}
```

```
public class Trailer extends Movie {
  /** Constructor: a trailer of movie t.
    Trailers are 1 minute long*/
  public Trailer(String t)
    {super(t, 1);}
}
```

```
public class Short extends Documentary {
  /** Constructor: instance with title t,
    length n, and topic p */
  public Short(String t, int n, String p)
    { super(t, n, p); }

  /** displays acknowledgement */
  public String showAck()
    {return "We thank our director";}

  /** = "Short Doc" */
  public String DocumentaryType()
    { return "Short Doc"; }
}
```

13

(Fall'05) **Question 4 (30 points)** For each pair of statements below, write the value of d after execution. If the statements lead to an error, write "BAD" and briefly explain the error. (The question continues on the next page.)

```
Documentary e=
        new Short("Man on Wire", 5, "Bio");
boolean d=
        "Short Doc" .equals(e.DocumentaryType());
```

14

(Fall'05) **Question 4 (30 points)** For each pair of statements below, write the value of d after execution. If the statements lead to an error, write "BAD" and briefly explain the error. (The question continues on the next page.)

```
Documentary e=
        new Short("Man on Wire", 5, "Bio");
boolean d=
        "Short Doc" .equals(e.DocumentaryType());
```

`True`. method equals here is from the string object

15

```
2.
Movie c=
    new Documentary(null, 3, "Carter Peace Center");

int d= c.popularity();
```

16

```
public class Movie {
  private String title; // title of movie
  private int length; // length in minutes
  /** Constructor: document with title t
      and len minutes long */
  public Movie(String t, int len) {
    title= t; length= len;
  }
  /** = title of this Movie */
  public String getTitle()
    { return title; }
  /** = length of document, in minutes */
  public int getLength()
    { return length; }
  /** = the popularity:
      shorter means more popular */
  public int popularity()
    { return 240 - length; }
}
```

```
public class Documentary extends Movie {
  private String topic; // …

  /** Constructor: instance with title t,
      length n, and topic p */
  public Documentary(String t, int n,
                     String p) {
    super(t, n);
    topic= p;
  }

  /** = "Documentary" */
  public String DocumentaryType()
    { return "Documentary"; }

  /** = popularity of this instance */
  public int popularity()
    { return 200 - getLength(); }
}
```

```
public class Trailer extends Movie {
  /** Constructor: a trailer of movie t.
    Trailers are 1 minute long*/
  public Trailer(String t)
    {super(t, 1);}
}
```

```
public class Short extends Documentary {
  /** Constructor: instance with title t,
    length n, and topic p */
  public Short(String t, int n, String p)
    { super(t, n, p); }

  /** displays acknowledgement */
  public String showAck()
    {return "We thank our director";}

  /** = "Short Doc" */
  public String DocumentaryType()
    { return "Short Doc"; }
}
```

17

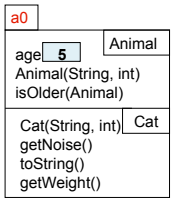
QUESTION: Which method is called by Animal t= new Cat("A",5); t.toString() ?

A. the one in the hidden partition for Object of a0

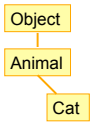
B. the one in partition Animal of a0

C. the one in partition Cat of a0

D. None of these

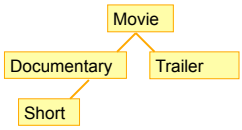


the class hierarchy:



18

```
2.
Movie c=
    new Documentary(null, 3, "Carter Peace Center");
int d= c.popularity();
```



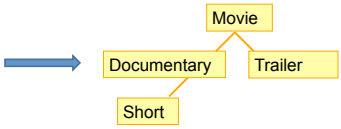
- What is the apparent class?
- **Answer: 197.** method popularity of class Documentary is called

19

```
3.
Short b= (Short)(new Documentary("", 2, "WMD"));
int d= b.DocumentaryType().length();
```

20

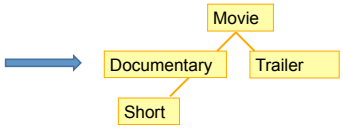
```
3.
Short b= (Short)(new Documentary("", 2, "WMD"));
int d= b.DocumentaryType().length();
```



- From documentary, can go (cast) up and back down to documentary.
- Think what would happen for the call b.showAck()

21

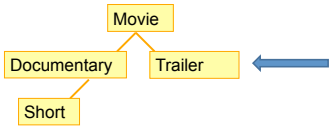
```
3.
Short b= (Short)(new Documentary("", 2, "WMD"));
int d= b.DocumentaryType().length();
```



- From documentary, can go (cast) up and back down to documentary.
- Think what would happen for the call b.showAck().
- Answer: BAD

22

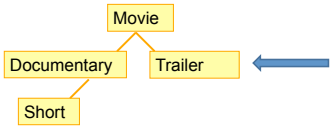
```
4.
Movie a= (Movie)(new Trailer("Harry Potter"));
int d= a.popularity();
```



- **The cast is legal!**
- **Which popularity() method is called?**

23

```
4.
Movie a= (Movie)(new Trailer("Harry Potter"));
int d= a.popularity();
```



- **The cast is legal!**
- Method popularity() from Trailer is called (inherited by Trailer)

24

```
5.
Movie f= new Short("War", 1, "Vietnam");
char d= f.DocumentaryType().charAt(1);
```

The methods that can be called are determined by the apparent type:

Only components in the apparent class (and above)!!!


```
5.
Movie f= new Short("War", 1, "Vietnam");
char d= f.DocumentaryType().charAt(1);
```

The methods that can be called are determined by the apparent type:

Only components in the apparent class (and above)!!!

f.DocumentaryType() is illegal. Syntax error.

Answer: BAD

Recap: equals(Object ob)

- In class Object
  - b.equals(d) is the same as b == d
    - Unless b == null (why?)
- Most of the time, we want to use *equals* to compare fields. We need to override this method for this purpose

(Fall'05) Question 4 (24 points). (a) Write an instance method equals (Object obj) for class Documentary

```
public class Documentary extends Movie {
/** = "obj is a Documentary with the same values
      in its fields as this Documentary" */
public boolean equals(Object obj) {




}
```



```
public class Documentary extends Movie {
/** = "obj is a Documentary with the same values
      in its fields as this Documentary" */
public boolean equals(Object obj) {

  if (!(obj instanceof Documentary) {

  }




}
```



```
public class Documentary extends Movie {
/** = "obj is a Documentary with the same values
      in its fields as this Documentary" */
public boolean equals(Object obj) {

  if (!(obj instanceof Documentary) {
      return false;
  }




}
```

```
public class Documentary extends Movie {
/** = "obj is a Documentary with the same values
      in its fields as this Documentary" */
public boolean equals(Object obj) {

  if (!(obj instanceof Documentary) {
      return false;
  }
  Documentary docObj= (Documentary)obj;

}
```

Don't forget to cast.
This is a legal cast. (Why?)

31

```
public class Documentary extends Movie {
/** = "obj is a Documentary with the same values
      in its fields as this Documentary" */
public boolean equals(Object obj) {

  if (!(obj instanceof Documentary) {
      return false;
  }
  Documentary docObj= (Documentary)obj;
  return
    getTitle().equals(docObj.getTitle()) &&
    getLength() == docObj.getLength() &&
    topic.equals(docObj.topic);
```

32

Let's capture the essence of animals

```
/** representation of an animal */
public class Animal {
  private int birthDate; // animal's birth date
  private String predator; // predator of this animal
  private String prey; // class of animals this hunts
  …
  // move the animal to direction…
  public void move(…){
      …
  }
  // make the animal eat…
  public void eat (…){
      …
  }
  …
}
```

33

Problems

- Animal is an abstract concept
  - Creating an abstract animal doesn't make sense in the real world
  - Dogs, cats, snakes, birds, lizards, all of which are animals, must have a way to eat so as to get energy to move
- However…
  - Class Animal allows us to create a UFA (unidentified flying animal), i.e. instance of Animal
  - If we extend the class to create a real animal, nothing prevent us from creating a horse that doesn't move or eat.

34

Solutions

- How to prevent one from creating a UFA?
  - Make class Animal abstract
    - **Class cannot be instantiated**
  - How? Put in keyword **abstract**
- How to prevent creation paralyzed dogs or starving sharks?
  - Make the methods move and eat abstract
    - **Method must be overridden**
  - How? Put in keyword **abstract** and replace the body with ";"

35

Making things abstract

```
/** representation of an animal */
public abstract class Animal{
  private int birthDate; // birth date
  private String predator; // animal's predator
  private String prey; // What animal hunts
  …
  // Move the animal move in direction  …
  public abstract void move(…);

  // Make the animal eat…
  public abstract void eat (…);
}
```

36

Array: object

Can hold a fixed number of values of the **same type**.

The **type** of the array:

- int[]
- String[]
- Integer[]

Basic form of a declaration: **int[] x**

Does not create array, it only declares x. x's initial value is **null**.

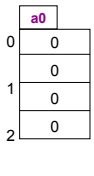
x **null** int[]

Array creation: **new int**[4]

Assignement: **int**[] t= **new int**[4]

t **a0** int[]

Elements of array are numbered: 0, 1, 2, …, x.length–1;



37

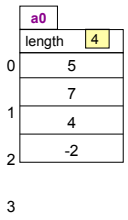
Array: length

Array length: an instance field of the array.

This is why we write x.length, not x.length( )

Length field is **final:** cannot be changed.

Length remains the same once the array has been created.



The length is not part of the array type.

The type is **int[]**

An array variable can be assigned arrays of different lengths.

```
int[] x;
x= new int[4];
x= new int[32];
```

38

int[] x ;

x **null** int[]

x= **new int**[4];

Create array object of length 4, store its name in x

x **a0** int[]

x[2]= 5;
x[0]= -4;

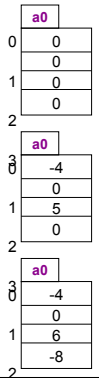
Assign 5 to array element 2 and -4 to array element 0

x[2] is a reference to element number 2 of array x

int k= 3;
x[k]= 2* x[0];
x[k-1]= 6;

Assign 2*x[0], i.e. -8, to x[3]
Assign 6 to x[2]

Arrays



39

Difference between Vector and array

Declaration:	**int[]** a;	Vector v;
	Elements of a: **int** values	Elements of v: any Objects
Creation:	a= **new int**[n];	v= **new** Vector();
	Array always has n elements	Number of elements can change
Reference:	a[e]	v.get(e)
Change element:	a[e]= e1;	v.set(e, e1);

Array locations a[0], a[1], … in successive locations in memory. Access takes same time no matter which one you reference.

Elements all the same type (a primitive type or class type)

Can't tell how Vectors are stored in memory. Referencing and changing elements done through method calls

Elements of any Object type (but not a primitive type). Casting may be necessary when an element is retrieved.

Array initializers

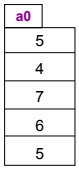
Instead of

int[] c= **new** int[5];

c[0]= 5; c[1]= 4; c[2]= 7; c[3]= 6; c[4]= 5;

Use an array initializer:

int[] c= **new int**[] {5, 4, 7, 6, 5};



array initializer: values must have the same type, in this case, **int**. Length of the array is the number of values in the list

41

Question 3 (20 points)

a) Consider the program segment below. Draw all variables (with their respective values) and objects created by execution of this program segment.

```
int[] z= new int[] {3, 2, 1};
String[] s= new String[2];
z= new int[2];
```

b) Give an expression to reference the second element of z.

c) What is the result of the expression `s[1].length()` after the execution of the code above?

d) Give the declaration of a single variable v to store the values "1" and "Hi" at the same time.

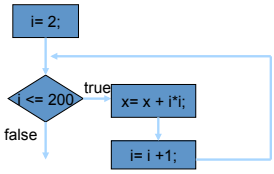
42

Execution of the for-loop

The for-loop:

```
for (int i= 2; i <= 200; i= i +1) {
    x= x + i*i;
}
```

loop counter: i
initialization: **int** i= 2;
loop condition: i <= 200;
increment: i= i + 1
repetend or body: { x= x + i; }



To execute the for-loop.
1. Execute initialization.
2. If loop condition false, terminate execution.
3. Execute repetend.
4. Execute increment, repeat from step 2.

Called a "flow chart"

Note on ranges.

2..5 contains 2, 3, 4, 5. It contains 5+1 – 2 = 4 values
2..4 contains 2, 3, 4. It contains 4+1 – 2 = 4 values
2..3 contains 2, 3. It contains 3+1 – 2 = 2 values
2..2 contains 2. It contains 2+1 – 2 = 1 values

The number of values in **m**..**n** is **n**+1 – **m**.

2..1 contains . It contains 1+1 – 2 = 0 values

3..1 contains . This is an invalid range!

In the notation m..n, we require always, without saying it, that
m <= n + 1 .
If m = n + 1, the range has 0 values.

Invariants

- **Assertions**: true-false statements (comments) asserting your beliefs about (the current state of) your program.
  // x is the sum of 1..n <- asserts a specific relationship between x and n
- **Invariant:** an assertion about the variables that is true before and after each iteration (execution of the repetend).

Finding an invariant

```
// Store in double variable v the sum
//   1/1 + 1/2 + 1/3 + 1/4 + 1/5 + … + 1/n

 v= 0;
 // invariant: v = sum of 1/i for i in 1..k-1
for (int k= 1; k <= n;  k= k +1) {
        Process k;
}
// v =1/1 + 1/2 + … + 1/n
```

Command to do something and

equivalent postcondition

The increment is executed after the repetend and before the next iteration

What is the invariant?

1 2 3 … k-1 k k+1 … n