

Lecture Set 5: Design and Classes

This Set:

- Methods and Parameter Passing
- Basics of program design
- Pseudo-code
- Objects and classes
- Heaps
- Garbage Collection
- More about Creating Objects and classes in Java
- Methods
- Constructors, Accessors, Mutators
- Equality
- Printing an object

CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorri)



methods: defining and invoking

- One useful type of method for this project:
 - defined within the FlagMaker class
 - invoked from within another method that is defined in the same class
 - we'll do a lot more variations later
- Defined based on a name and a list of parameters

```
public static void name(parameterlist) {  
    body  
}
```
- Invoked by stating its name and giving an argument for each element of the parameter list

```
name(argumentlist);
```

CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorri)

1



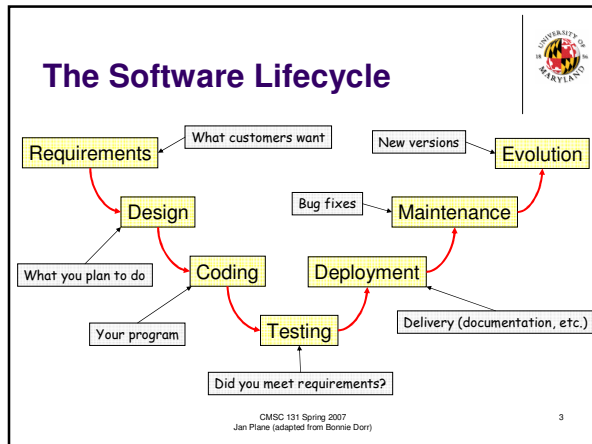
method information: parameters and arguments

- parameter list
 - type name for each item in the list
 - e.g. (MyGrid grid, char where)
- argument list
 - expression for each item in the list
 - e.g. (grid, 't')
- Matched between the arguments and the parameters based on position in the list

CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorri)

2





In the Real World, Requirements and Design Rule

- Getting requirements right is essential for successful projects
 - FBI electronic case file (junked after \$180m)
 - IRS system upgrade in late 90s (junked after >\$2bn)
 - FAA air-traffic control (false starts, >\$10bn spent)
- Good design makes other parts of lifecycle easier
- In “the real world” coding typically < 30% of total project costs
- A good design improves:
 - efficiency (speed)
 - efficiency (memory)
 - ease of coding
 - ease of debugging
 - ease of expansion

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

Program Design

- There are many aspects to good design
 - Architecture
 - Modeling
 - Requirements decomposition
 - **Pseudo-code**
- In this class we will focus on latter

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

What Is “Pseudo-code”?



- When developing a complex part of a program (an algorithm), one of the tools often useful is pseudo-code.
- It's not English, not programming language -- somewhere between.
- Captures the flow of the program without worrying about language-specific details.

CMSC 131 Spring 2007
Jan Plare (adapted from Bonnie Dorri)

6

Example:



- **Requirement:** email program that allows you to send a message either to one person, or to your whole address book
- **Pseudo-code:**

```
prompt "Enter message: "  
input message  
prompt "Send to whole address book? "  
input answer  
if answer == "no"  
    prompt "Enter recipient: "  
    input recipient  
    send message to recipient  
otherwise  
    for each recipient, r, in address book  
        send message to r
```

CMSC 131 Spring 2007
Jan Plare (adapted from Bonnie Dorri)

7

What Is Pseudo-Code? (cont.)



- NOT English
- NOT a program
- Something in-between
 - Captures the "logic" and "flow" of the algorithm
 - Note that pseudo-code could be translated into ANY programming language (not just Java)
- Good programming practice
 - Write pseudo-code first and keep it as your design
 - Include it as comments in your code to help you connect code to design

CMSC 131 Spring 2007
Jan Plare (adapted from Bonnie Dorri)

8

Testing



- Some testing is done by customer (**acceptance testing**)
 - E.g. testing we do on your projects!
 - You want to avoid errors surfacing during acceptance testing
- How to avoid errors during acceptance testing?
 - Test thoroughly before release
 - Cover all cases in code (if/else branches, etc.)
 - Identify "corner cases" (extreme values of inputs) and test with these
- We will study testing more later in semester

CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorr)

9

Questions



- What is `System.in` in `System.out.println()`?
- Why use `str.equals("cat")` to compare equality of `String str` and `"cat"`?
- Is the similarity of the notations
`System.out.println()`
`str.equals()`
`sc.nextInt()`
important, or coincidental?

CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorr)

10

Objects



- Bundles of (related)
 - **data** ("state")
 - **operations** ("behavior")
- Data often referred to as **instance variables**
- Operations usually called **methods**
- Invoking operations can change state (values stored in instance variables)

CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorr)

11

Sample Student Object



State		Methods	
Name	Kerry Keenan	getAge	date → age
ID	444230695	getGrades	semester → grades
DOB	06-22-1987		etc.
Major	CMSC		
	etc.		

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

12

Accessing State / Methods



- If
 - `o` is an object
 - `v` is an instance variable of the object
 - `m` is a method of the object
- Then
 - `o.v` is how to access the data `v` in `o`
 - `o.m` is how to invoke `m` in `o`
- So
 - `System` is an object, with `out` an instance variable
 - `out` is also an object, with `println` a method
 - `System.out.println` is how to access this method!
- Suppose `str` is a String
 - `str` is an object!
 - Methods of this object: `equals`, `compareTo`, etc.
 - `str.equals`, `str.compareTo`, etc. invokes these methods on that object

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

13

Object-Oriented Programming



- Programs are collections of interacting objects
- Writing programs involves identifying what the objects should be and programming them
- Object-oriented languages provide features to ease object-oriented programming
- Defining objects involves identifying
 - state
 - methods

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

14

Classes



- “Blueprints” (“templates”) for objects
- Classes include specifications of
 - Instance variables (including types, etc.) to include in objects
 - Implementations of methods to include in objects
- Classes can include other information also, as will be seen later
 - static methods / instance variables
 - public / private methods, instance variables
 - And so on

CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorr)

15

Student Class Example



Conceptually:

- Instance variables:
 - String name
 - int ID
 - int dateOfBirth
 - String major
- Methods
 - getAge
 - getGrades
 - etc.
- The actual class implementation will include code for the methods
- This describes a blueprint for student objects

CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorr)

16

How Are Objects Created?



- In Java: using `new`
Recall:
`Scanner sc = new Scanner (System.in);`
- Invoking `new`:
 - creates fresh copies of instance variables in the “heap”
 - returns the “address” where the fresh variables are stored
- Heap? Address?

CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorr)

17

Heap = “Fresh Memory”



- While a program is running, some memory is used to store variables
 - Terminology: **stack**
 - We have been representing stack as table, e.g.

Variable	Value
x	3
y	4, 5

- Rest of memory is called **heap** and can be used for other purposes, including storing new objects

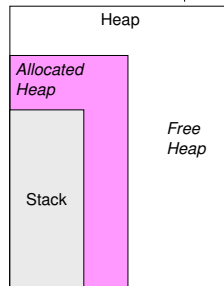
CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorri)

18

Main Memory



- Stack grows, shrinks during program execution (why?)
- So does “allocated heap” (part of heap in use)
- Unallocated part of heap is called “free”



CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorri)

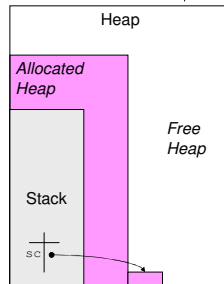
19

Object Creation



- New space allocated in heap to store instance variables
- **Reference** (= address) to this space is returned

```
Scanner sc = new (...);
```



CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorri)

20

Strings Are Objects



- Where is new in

```
String name = "Narita"; ?
```

- Java provides it!

- String is special because it is used so often
- Java automatically "fills in" new
- You can too:

```
String name = new String("Narita");
```

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

21

In Java, 9 Sorts of Variables



- 8 primitively typed

- Types are the 8 built-ins (int, byte, double, etc.)
- Storage allocated on stack based on type
- Value stored in stack
- e.g. int x

- Reference typed

- Types are classes
- Storage allocated on stack to hold one memory address (typically, one word)
- What is stored in stack is reference to heap, where actual data is stored
- e.g. Scanner sc = new Scanner (System.in);

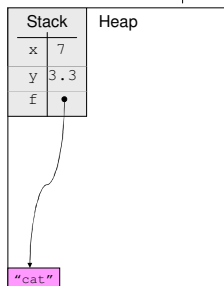
CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

22

Example



```
int x = 7;  
float y = 3.3;  
String f = "cat";
```



CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

23

Heap Issues

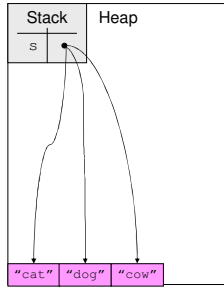
- What happens if `new` is called and there is no free heap?

Crash!

- What happens if following are executed?

```
String s;  
s = new String("cat");  
s = new String("dog");  
s = new String("cow");
```

- Wasted heap
 - "cat", "dog" no longer referenced by stack
 - Crashes become a problem!



CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

24

Garbage Collection

- This "heap management" or "memory management" issue is central in CS
- Java copes by invoking **garbage collector** to reclaim unused but still-allocated heap space
- Garbage collector **reclaims** memory in allocated heap and returns it to free heap
- In previous example, "cat" and "dog" would be reclaimed

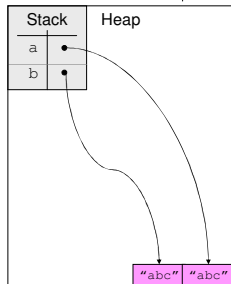
CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

25

Example

```
String a = new String("abc");  
String b = new String("abc");  
if (a == b) {  
    println("Equal");  
} else {  
    println("Not equal");  
}
```

- Not equal is printed



CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

26

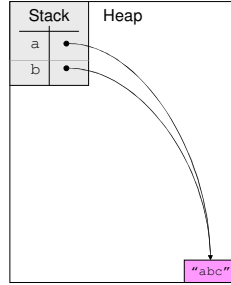
Contrasting Example

```
String a = new String("abc");
```

```
String b = a;
```

```
if (a == b){  
    println ("Equal");  
} else {  
    println ("Not equal");  
}
```

- Equal is printed
- This is called ALIASING: Two variables refer to same object.
- Can be DANGEROUS!!
- What if we really want to make a copy?
String a = "abc"
String b = new String(a);



CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorri)

27

"equals"

- == checks if two reference variables refer to the same object
- Methods like `str.equals()` check if two different objects have the same "content"
- Other classes will have an `equals` method also

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorri)

28

Classes in Java

- Class declarations have the following form in Java:

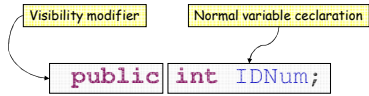
```
Visibility modifier: more later in class  
class keyword  
class name  
public class Student {  
    class body: instance variables, methods  
}
```

- When you create a class in Eclipse, it generates this template for you

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorri)

29

Anatomy of an Instance Variable Declaration



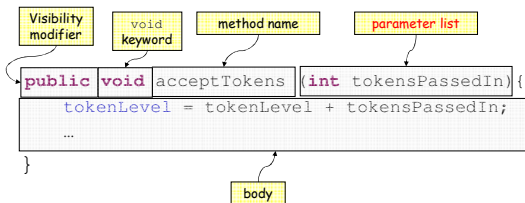
CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorr)

30

Anatomy of a Method Declaration (1)



... for methods that do not return values



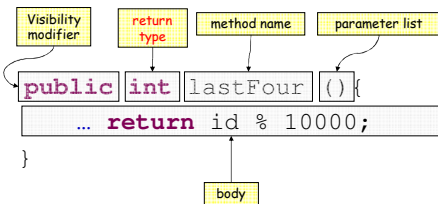
CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorr)

31

Anatomy of a Method Declaration (2)



... for methods that return values



CMSC 131 Spring 2007
Jan Plante (adapted from Bonnie Dorr)

32

Return Type



- Methods that return values must specify the type of the value to be returned
- The bodies of these methods use `return` to indicate when a value is to be returned
- The value being returned must have the same type as the return type

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

33

Object Creation



- Once a class is defined, objects based on that class can be created using `new`:
`new Student ()`
- To assign an object to a variable, the variable's type must be the class of the object
`Student s = new Student ();`
- Each object has its own copies of all the instance variables in the class (except for certain kinds we'll study later)
- Instance variables and methods in an object can be accessed using `."` or using setter (mutator) methods
`s.IDNum = 123456789;`
`s.setIDNum(123456789);`

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

34

Constructors



- Special "methods" in class definitions to specify how objects are created
- Form of a constructor definition:

```
Student (String nameDesired, int IDDesired, int tokensDesired) {  
    name = nameDesired;  
    id = IDDesired;  
    tokenLevel = tokensDesired;  
}
```
- Can have more than one constructor, provided argument lists are different

```
Student (int IDDesired) {  
    id = IDDesired;  
}
```
- Java includes *default* constructor (no arguments), which you can redefine (*override*)

```
Student () {  
    tokenLevel = 3;  
}
```

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

35

Equality Testing



- Need to defined what it means for two students to be equal

```
public boolean equals (Student otherStudent){
    if (otherStudent == NULL){
        return false;
    }else if (id == otherStudent.id){
        return true;
    }else{
        return false;
    }
}
```

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

36

Objects to Strings



- What happens if we try to print a Student object?
 - invoke `println` using a `Student` object as an argument?
`Student s1 = new Student ();`
`System.out.println (s1);`
- Something like this prints:
`Student@82ba41`

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

37

Java Knows “How” To Print Any Object



- Why?
 - Every class has a default `toString` method
 - `toString` converts objects into strings
 - `System.out.println` calls this method to print an object
 - Default: object type and address
- `toString` can be overridden!

// The method for converting Students to strings

```
public String toString () {
    return (name + ": " + id);
}
```

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorr)

38

Static Data Members and Static Methods



- Not contained in or associated with an object of that type
- Accessed by the `ClassName.variableName` or by `ClassName.methodName`
- rather than by `objectName.variableName` or by `objectName.methodName`

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorri)

39

Set / Get Methods



- We have been using `=` to modify instance variables and accessing variables directly to read values
- Generally, this is not good practice because it imposes restrictions on class implementation
- Better
 - `set` methods to set values (mutators)
 - `get` methods to read values (accessors)

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorri)

40

Set Methods (Mutators)



```
public void setID (int newID) {  
    id = newID;  
}
```

- Can also do consistency checking

```
public void setTokenLevel (int newTokenLevel) {  
    if (newTokenLevel <= 3) {  
        tokenLevel = newTokenLevel;  
    } else {  
        System.out.println ( "Bad argument to setTokenLevel: " + newTokenLevel);  
    }  
}
```

CMSC 131 Spring 2007
Jan Plane (adapted from Bonnie Dorri)

41

Get Methods (Accessors)



- Sole purpose is to return values of state

```
public int getID () {  
    return id;  
}
```

- Why use them?

- The state information may not always be stored in a single instance variable, since implementations may change
- You give designers option of changing instance variables

CMSC 131 Spring 2007
Jan Piatek (adapted from Bonnie Dorr)

42
