COMPUTER SCIENCE DEPARTMENT PICNIC

Welcome to the 2016-2017 Academic year !

Meet your faculty, department staff, and fellow students in a social setting. Food and drink will be provided.

When: Saturday, September 10th
Time: 11am – 2pm
Where: City Park Shelter #7

Operations

- Push the power button and hold.
- Once the light begins blinking, enter the room code
  - This room's code is BC
- When a question is asked, you have 30 seconds to respond
- Enter the letter of the appropriate answer
- When you enter the letter of the answer, your i-clicker will blink green.
- It is your responsibility to check for that green light.

I Forgot…

- If you forgot your IClicker, or your batteries fail during the exam
  - Your worst quiz score is not counted to cover this situation.
  - All other quizzes count.
- If you have an excused absence, you may have the quiz score exempted.

Am I here?

A. Yes
B. No
C. Define here…

Java classes

Savitch, ch 5

Outline

- Objects, classes, and object-oriented programming
  - relationship between classes and objects
  - abstraction
- Anatomy of a class
  - instance variables
  - instance methods
  - constructors

Objects and classes

- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Writing programs that perform most of their behavior as interactions between objects.
- **class**: 1. A program. or,
  2. A blueprint of an object.
  - classes you may have used so far:
    `String`, `Scanner`, `File`
- We will write classes to define new types of objects.

7

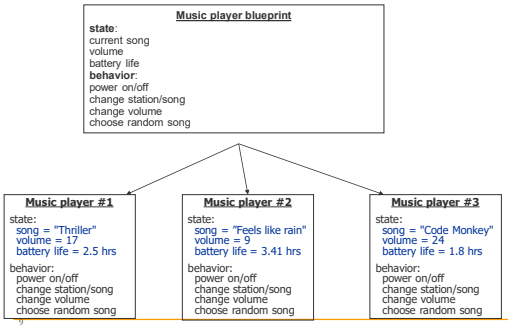
Abstraction

- **abstraction**: A distancing between ideas and details.
  - Objects in Java provide abstraction:
    We can use them without knowing how they work.

- You use abstraction every day.
  Example: Your portable music player.
  - You understand its external behavior (buttons, screen, etc.)
  - You don't understand its inner details (and you don't need to).

8

Class = blueprint, Object = instance



9

How often would you expect to get snake eyes?

If you're unsure on how to compute the probability then you write a program that simulates the process

10

Snake Eyes

```
public class SnakeEyes {
  public static void main(String [] args){
     int ROLLS = 100000;
     int count = 0;
     Die die1 = new Die();
     Die die2 = new Die();
     for (int i = 0; i < ROLLS; i++){
         if (die1.roll() == 1 && die2.roll() == 1){
            count++;
         }
     }
     System.out.println("snake eyes probability: " +
  count / ROLLS);
  }
}
```

Need to write the Die class!

11

Die object

- State (data) of a `Die` object:

Instance variable	Description
`numFaces`	the number of faces for a die
`faceValue`	the current value produced by rolling the die

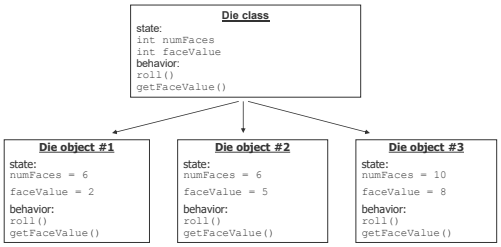
- Behavior (methods) of a `Die` object:

Method name	Description
`roll()`	roll the die
`getFaceValue()`	retrieve the value of the last roll

12

The Die class

- The class (blueprint) knows how to create objects.




```
Die die1 = new Die();
```

13

IC Question 2

- Why is abstraction a strength when we program?

A. It allows us to identify where we use classes
B. It allows us to use objects without knowing how they work
C. It allows us to use variables without knowing how they work
D. All of the above

14

IC Question 2 Answer

- Why is abstraction a strength when we program?

A. It allows us to identify where we use classes
B. It allows us to use objects without knowing how they work
C. It allows us to use variables without knowing how they work
D. All of the above

15

IC Question 3

- What is the range of possible values of the variable faceValue in the Die class?

A. 1..6
B. 1..8
C. 1..numFaces
D. 1..faceValue
E. None of the above

16

IC Question 3 Answer

- What is the range of possible values of the variable faceValue in the Die class?

A. 1..6
B. 1..8
C. 1..numFaces
D. 1..faceValue
E. None of the above

17

Object state: instance variables

Die class

- The following code creates a new class named `Die`.

```
public class Die {
    int numFaces;
    int faceValue;
}
```

declared outside of any method

  - Save this code into a file named `Die.java`.
- Each `Die` object contains two pieces of data:
  - an `int` named `numFaces`,
  - an `int` named `faceValue`
- No behavior (yet).

19

Instance variables

- **instance variable**: A variable inside an object that holds part of its state.
  - Each object has *its own copy*.
- Declaring an instance variable:

  <type> <name> ;

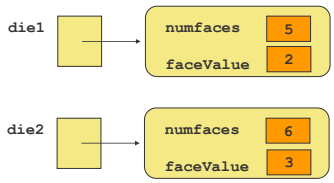
```
public class Die {
    int numFaces;
    int faceValue;
}
```

20

Instance variables

Each `Die` object maintains its own `numfaces` and `faceValue` variable, and thus its own state

```
Die die1 = new Die();
Die die2 = new Die();
```



21

Accessing instance variables

- Code in other classes can access your object's instance variables.
  - Accessing an instance variable: dot operator

    <variable name>* . *<instance variable>
  - Modifying an instance variable:

    <variable name>* . *<instance variable> = ***<value>*** ;
- Examples:

```
System.out.println("you rolled " + die.faceValue);
die.faceValue = 2;
```

22

Client code

- `Die.java` can be made executable by giving it a main …
  - We will almost always do this…. WHY?
  - To test the class Die before it is used by other classes
- or can be used by other programs stored in separate `.java` files.
  - **client code**: Code that uses a class

<u>Roll.java</u> (client code)

```
main(String[] args) {
    Die die1 = new Die();
    die1.numFaces = 6;
    die1.faceValue = 5;

    Die die2 = new Die();
    die2.numFaces = 10;
    die2.faceValue = 3;
    ...
}
```

<u>Die.java</u>

```
public class Die {
    int numFaces;
    int faceValue;
}
```

23

IC Question 4

- When modifying an instance variable by doing the following:

  xxx.yyy=2;

xxx is which of the following:

A. Class name
B. Variable name
C. Object name

24

IC Question 4 Answer

- When modifying an instance variable by doing the following:

  xxx.yyy=2;

xxx is which of the following:

A. Class name
B. Variable name
C. Object name

25

IC Question 5

- If we have defined a class to provide functionality to client code, what is the purpose of the main method is that class?

A. To provide a mechanism for unit testing
B. To provide print statements
C. To allow the programmer to build the class
D. None of the above

26

IC Question 5 Answer

- If we have defined a class to provide functionality to client code, what is the purpose of the main method is that class?

A. To provide a mechanism for unit testing
B. To provide print statements
C. To allow the programmer to build the class
D. None of the above

27

Object behavior: methods

OO Instance methods

- Classes combine state and behavior.
- **instance variables:** define state
- **instance methods**:
  define behavior for each object of a class.
  methods are the way objects communicate with each other and with users
- instance method declaration, general syntax:

```
public <type> <name> ( <parameter(s)> ) {
    <statement(s)> ;
}
```

29

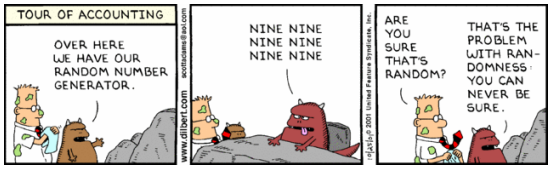
Rolling the dice: instance methods

```
public class Die {
   int numFaces;
   int faceValue;
   public int roll (){
      faceValue = (int)(Math.random() * numFaces) + 1;
      return faceValue;
   }
}
```

```
Die die1 = new Die();
die1.numFaces = 6;
int value1 = die1.roll();
Die die2 = new Die();
die2.numFaces = 10;
int value2 = die2.roll();
```

Think of each `Die` object as having its own copy of the `roll` method, which operates on that object's state

30



Object initialization: constructors

Initializing objects

- When we create a new object, we can assign values to all, or some of, its instance variables:

```
Die die1 = new Die(6);
Die die2 = new Die();
```

Die constructor

```
public class Die {
   int numFaces;
   int faceValue;
   public Die (int faces) {
      numFaces = faces;
      faceValue = 1;
   }
  // can't have both of these constructors at once
   public Die (int faceVal) {
    numFaces = 6;
    faceValue = faceVal;
   }
   public int roll (){
      faceValue = (int)(Math.random()*numFaces) + 1;
      return faceValue;
   }
}
```

```
Die die1 = new Die(6);
```

34

Constructors

- **constructor**: creates and initializes a new object

  public ***<type>*** (***<parameter(s)>***) {
  <statement(s)> ;
  }

  - For a constructor the <type> is the **name of the class**
  - A constructor runs when the client uses the `new` keyword.
  - A constructor implicitly returns the newly created and initialized object.
  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all the object's fields to 0 or null.

35

Multiple constructors are possible

```
public class Die {
   int numFaces;
   int faceValue;

   public Die () {
      numFaces = 6;
      faceValue = 1;
   }
  // OK to have both of these constructors at once
   public Die (int faces) {
      numFaces = faces;
      faceValue = 1;
   }
}
```

```
Die die1 = new Die(5);
Die die2 = new Die();
```

36

IC Question 2

```
public <type> ( <parameter(s)> ) {
    <statement(s)> ;
}
```

- For a constructor the <type> is which of the following:
  A. The return type
  B. The method type
  C. The name of the class
  D. The type of the parameter

37

IC Question 2 Answer

```
public <type> ( <parameter(s)> ) {
    <statement(s)> ;
}
```

- For a constructor the <type> is which of the following:
  A. The return type
  B. The method type
  C. The name of the class
  D. The type of the parameter

38

Encapsulation

Encapsulation

- **encapsulation**:
  Hiding implementation details of an object from clients.
- Encapsulation provides *abstraction*;
  we can use objects without knowing how they work.
  The object has:
  - an external view (its behavior)
  - an internal view (the state and methods that accomplish the behavior)

40

Implementing encapsulation

- Instance variables can be declared *private* to indicate that no code outside their own class can access or change them.
  - Declaring a private instance variable:
    `private <type> <name> ;`
  - Examples:

```
private int faceValue;
private String name;
```

- Once instance variables are private, client code cannot access them:

```
Roll.java:11: faceValue has private access in Die
System.out.println("faceValue is " + die.faceValue);
                                        ^
```

41

Instance variables, encapsulation and access

- In our previous implementation of the Die class we used the public access modifier:

```
public class Die {
  public int numFaces;
  public int faceValue;
}
```

- We can encapsulate the instance variables using private:

```
public class Die {
  private int numFaces;
  private int faceValue;
}
```

But how does a client class now get to these?

42

Accessors and mutators

- We provide accessor methods to examine their values:

```
public int getFaceValue() {
    return faceValue;
}
```

  - This gives clients read-only access to the object's fields.
  - Client code will look like this:

```
System.out.println("faceValue is " + die.getFaceValue());
```

- **If required**, we can also provide mutator methods:

```
public void setFaceValue(int value) {
    faceValue = value;
}
```

Often not needed. Do we need a mutator method in this case?

43

Benefits of encapsulation

- Protects an object from unwanted access by clients.
  - Example: If we write a program to manage users' bank accounts, we don't want a malicious client program to be able to arbitrarily change a `BankAccount` object's balance.
- Allows you to change the class implementation later.
- As a general rule, all instance data should be modified only by the object, i.e. **instance variables should be declared private**

44

Access Protection: Summary

Access protection has three main benefits:

- It allows you to enforce constraints on an object's state.
- It provides a simpler client interface. Client programmers don't need to know everything that's in the class, only the public parts.
- It separates interface from implementation, allowing them to vary independently.

45

General guidelines

As a rule of thumb:

- Classes are public.
- Instance variables are private.
- Constructors are public.
- Getter and setter/mutator methods are public
- Other methods must be decided on a case-by-case basis.

46

IC Question 3

- Instance variables can be declared *which of the following* to indicate that no code outside their own class can access or change them.
  A. Public
  B. Instance
  C. Class
  D. Private
  E. None of the above

47

IC Question 3 Answer

- Instance variables can be declared *which of the following* to indicate that no code outside their own class can access or change them.
  A. Public
  B. Instance
  C. Class
  D. Private
  E. None of the above

48

Printing Objects

- We would like to be able to print a Java object like this:

```
Student student = new Student(…);
System.out.println("student: " + student);
```

- Would like this provide output that is more useful than what Java provides by default.
  - Need to provide a toString() method

49

The `toString()` method

- tells Java how to convert an object into a `String`
- called when an object is printed or concatenated to a `String`:

```
Point p = new Point(7, 2);
System.out.println("p: " + p);
```

  - Same as:

```
System.out.println("p: " + p.toString());
```

- Every class has a `toString()`, even if it isn't in your code.
  - The default is the class's name and a hex (base-16) hash-code:

```
Point@9e8c34
```

50

`toString()` implementation

```
public String toString() {
    code that returns a suitable String;
}
```

  - Example: toString() method for our Student class:

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

51

IC Question 4

- Every class must specify a `toString()`

A. `True`
B. `False`

IC Question 4 Answer

- Every class must specify a `toString()`

A. `True`
B. `False - classes have a default`

The default is the class's name and a hex (base-16) hash-code:

`Point@9e8c34`

Variable shadowing

- An instance method parameter can have the same name as one of the object's instance variables:

```
public class Point {
  private int x;
  private int y;
     …
  // this is legal
  public void setLocation(int x, int y) {
    // when using x and y you get the parameters

  }
```

  - Instance variables `x` and `y` are *shadowed* by parameters with same names.

Avoiding variable shadowing

```
public class Point {
    private int x;
    private int y;
    ...
    public void setLocation(int x_value, int y_value) {
        x = x_value;
        y = y_value;
    }
}
```

55

Avoiding shadowing using `this`

```
public class Point {
    private int x;
    private int y;
    ...
    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

- Inside the `setLocation` method,
  - When `this.x` is seen, the *instance variable* `x` is used.
  - When `x` is seen, the *parameter* `x` is used.

56

Multiple constructors

- It is legal to have more than one constructor in a class.
  - The constructors must accept different parameters.

```
public class Point {
    private int x;
    private int y;

    public Point() {
        x = 0;
        y = 0;
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
```

57

Constructors and this

- One constructor can call another using `this`:

```
public class Point {
    private int x;
    private int y;

    public Point() {
        this(0, 0);  //calls the (x, y) constructor
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```

58

Summary of this

- **`this`** : A reference to the current instance of a given class
- using `this`:
  - To refer to an instance variable:
    `this.`**variable**
  - To call a method:
    `this.`**method**`(`**parameters**`);`
  - To call a constructor from another constructor:
    `this(`**parameters**`);`

59

Example of using this

```
public class MyThisTest {
  private int a;
  public MyThisTest() {
    this(42);
  }
  public MyThisTest(int a) {
    this.a = a;
  }
  public void someSomething() {
    int a = 1;
    System.out.println(a);
    System.out.println(this.a);
    System.out.println(this);
  }
  public String toString() {
    return "MyThisTest a=" + a; // refers to the instance variable a
  }
}
```

60

IC Question 5

- **When you see this used like below, what is occurring?**

```
this(parameters);
```

A. Referring to an instance variable
B. Calling a method
C. Calling a constructor from another constructor
D. Calling a static method

61

IC Question 5 Answer

- **When you see this used like below, what is occurring?**

```
this(parameters);
```

A. Referring to an instance variable
B. Calling a method
C. Calling a constructor from another constructor
D. Calling a static method

62

Method overloading

- Can you write different methods that have the same name?
- Yes!

```
System.out.println("I can handle strings");
System.out.println(2 + 2);
System.out.println(3.14);
System.out.println(object);
Math.max(10, 15);        // returns integer
Math.max(10.0, 15.0);    // returns double
```

Useful when you need to perform the same operation on different kinds of data.

63

Method overloading

```
public int sum(int num1, int num2){
   return num1 + num2;
}
public int sum(int num1, int num2, int num3){
   return num1 + num2 + num3;
}
```

- A method's name + number, type, and order of its parameters: **method signature**
- The compiler uses a method's signature to bind a method invocation to the appropriate definition

64

The return value is not part of the signature

- You **cannot** overload on the basis of the return type (because it can be ignored)

Example of invalid overloading:

```
public int convert(int value) {
  return 2 * value;
}
public double convert(int value) {
  return 2.54 * value;
}
```

65

Example

- Consider the class Pet

```
class Pet {
   private String name;
   private int age;
   private double weight;

...
}
```

66

Example (cont)

```
public Pet()
public Pet(String name, int age, double weight)
public Pet(int age)
public Pet(double weight)
```

Suppose you have a horse that weights 750 pounds then you use:

```
Pet myHorse = new Pet(750.0);
```

but what happens if you do:

```
Pet myHorse = new Pet(750);  ?
```

67

IC Question 6

- The following is legal method overloading

public int add (int x, int y)
public double add (int a, int b)

A. True
B. False

68

IC Question 6 Answer

- The following is legal method overloading

public int add (int x, int y)
public double add (int a, int b)

A. True
B. False

69

Primitive Equality

- Suppose we have two integers `i` and `j`
- How does the statement `i==j` behave?
- `i==j` if `i` and `j` contain the same value

70

Object Equality

- Suppose we have two pet instances `pet1` and `pet2`
- How does the statement `pet1==pet2` behave?

71

Object Equality

- Suppose we have two pet instances `pet1` and `pet2`
- How does the statement `pet1==pet2` behave?
- `pet1==pet2` is true if ***both*** refer to the ***same*** object
- The `==` operator checks if the ***addresses*** of the two objects are equal
- May not be what we want!

72

Object Equality - extended

- If you want a different notion of equality define your own `.equals()` method.
- Use `pet1.equals(pet2)` instead of `pet1==pet2`
- The default definition of `.equals()` is the value of ==

  but for Strings the contents are compared

73

.equals for the Pet class

```
public boolean equals (Object other) {
  if (!other instanceof Pet) {
      return false;
  }
  Pet otherPet = (Pet) other;
  return ((this.age == otherPet.age)
    &&(Math.abs(this.weight - otherPet.weight) < 1e-8)
    &&(this.name.equals(otherPet.name)));
}
```

This is not explained correctly in the book (section 5.3)!!
// SHOW ECLIPSE EXAMPLE OF Equals code.

74

Naming things

- Computer programs are written to be read by humans and only incidentally by computers.
- Use names that convey meaning
- Loop indices are often a single character (i, j, k), but others should be more informative.
- Importance of a name depends on its scope: Names with a “short life” need not be as informative as those with a “long life”
- Read code and see how others do it

75