

0608 – Creating Classes

ICSI 201

Introduction to Computer Science

Qi Wang

Department of Computer Science

University at Albany

State University of New York

Course Materials

- All course materials are **copyrighted** and can be used by the students who are enrolled in the class only.
- Posting/sharing the course materials for other uses without permission is prohibited.

Outline

- Objects and Classes
- Anatomy of a Class
 - Class components
 - Design a class
 - Write code for a class
- Data Scope
- Encapsulation
- Overloading Methods and Constructors
- Static members of a class

Objects and Classes

Objects and Classes

- An object exists in memory, and performs a specific task.
- Objects have two general capabilities:
 - Objects can store data. The pieces of data stored in an object are known as *fields(instance variables/state)*.
 - Objects can perform operations. The operations that an object can perform are known as *methods(instance methods/behaviors)*.

Objects and Classes

- Consider a six-sided die (singular of dice).
 - Its **state** can be defined as which face is showing.
 - Its primary **behavior** is that it can be rolled.
- We can represent a die in software by designing a class called `Die` that models this state and behavior.
 - The class serves as the blueprint for a die object.
- We can then instantiate as many die objects as we need for any particular program.

Writing Classes

- The class that contains the `main` method is just the starting point of a program.
- True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality.
- An object has *state* and *behavior*.

Objects and Classes

- You have already used the following objects:
 - `Scanner` objects, for reading input
 - `Random` objects, for generating random numbers
 - `PrintWriter` objects, for writing data to files
- When a program needs the services of a particular type of object, it creates that object in memory, and then calls that object's methods as necessary.

Objects and Classes

- Classes: Where Objects Come From.
- A *class* is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods).
- You can think of a class as a code "blueprint" that can be used to create a particular type of object.

Objects and Classes

- When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed.
- Each object that is created from a class is called an *instance* of the class.

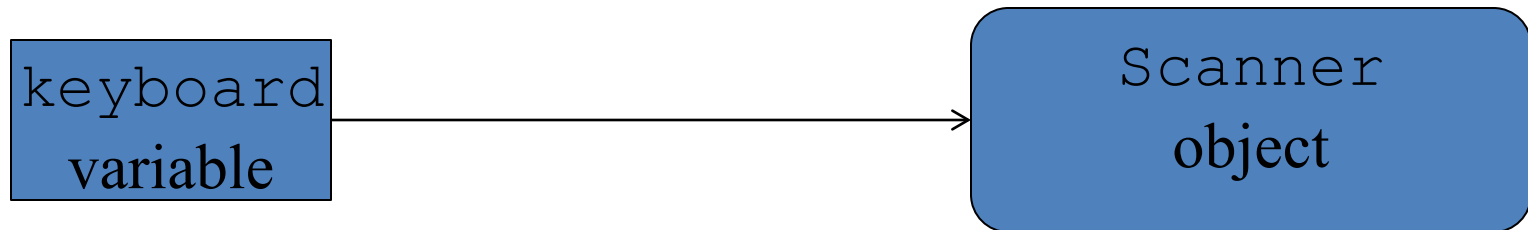
Objects and Classes

Example:

This expression creates a
Scanner object in memory.

```
Scanner keyboard = new Scanner(System.in);
```

The object's memory address
is assigned to the keyboard
variable.

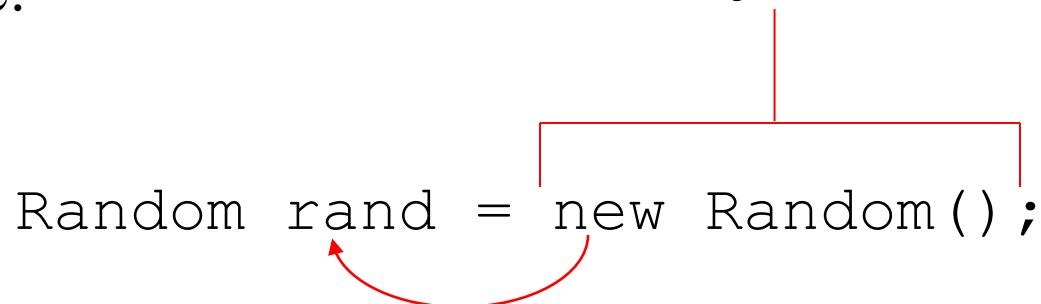


Objects and Classes

Example:

This expression creates a
Random object in memory.

```
Random rand = new Random();
```



The object's memory address is
assigned to the `rand` variable.



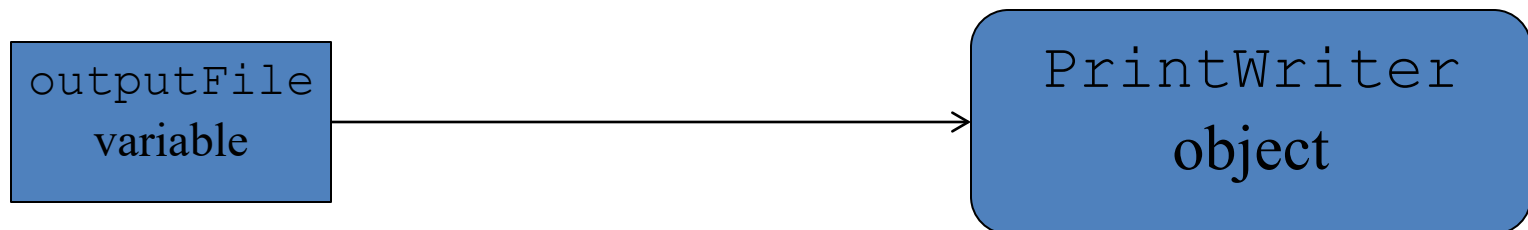
Objects and Classes

Example:

This expression creates a
PrintWriter object in memory.

```
PrintWriter outputFile = new PrintWriter("numbers.txt");
```

The object's memory address is assigned to
the `outputFile` variable.

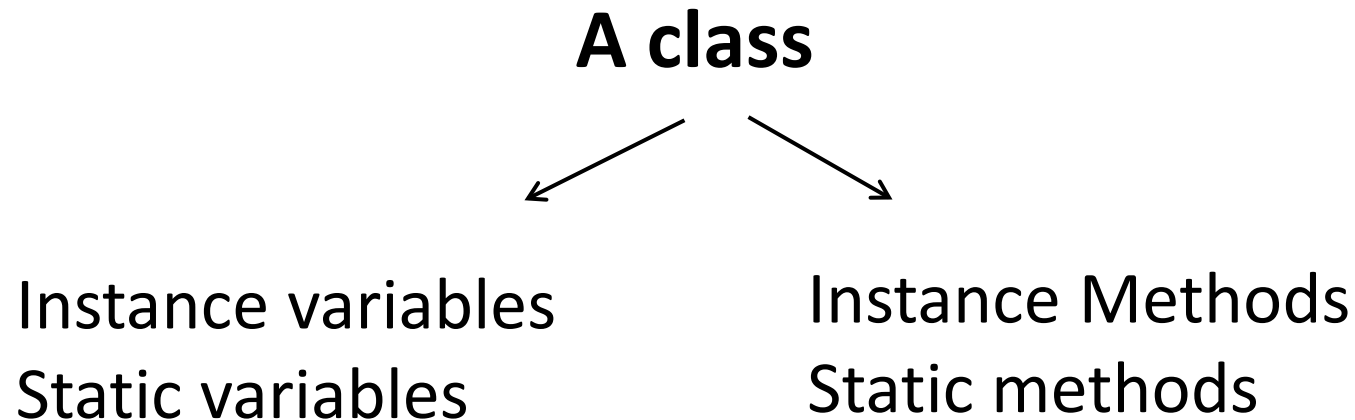


Objects and Classes

- The Java API provides many classes
 - So far, the classes that you have created objects from are provided by the Java API.
 - Examples:
 - `Scanner`
 - `Random`
 - `PrintWriter`
- We can write a class that models a type of object in real world, such as a die, a dog, or a student.

Anatomy of a Class

A class can contain instance variables, static variables, instance methods, and static methods.



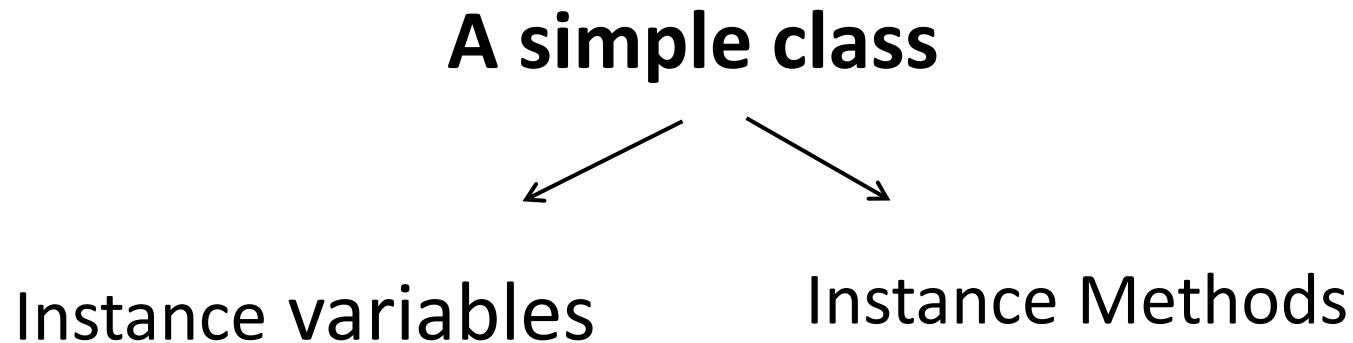
Instance vs. Static

- Each **data** owned by/associated with an instance is declared as an **instance variable**.
- Each **behavior** controlled by/associated with an instance is modeled in an **instance method**.
- Each **data** owned by/associated with a class is declared as a **static variable**.
- Each **behavior** controlled by/associated with a class is modeled in a **static method**.

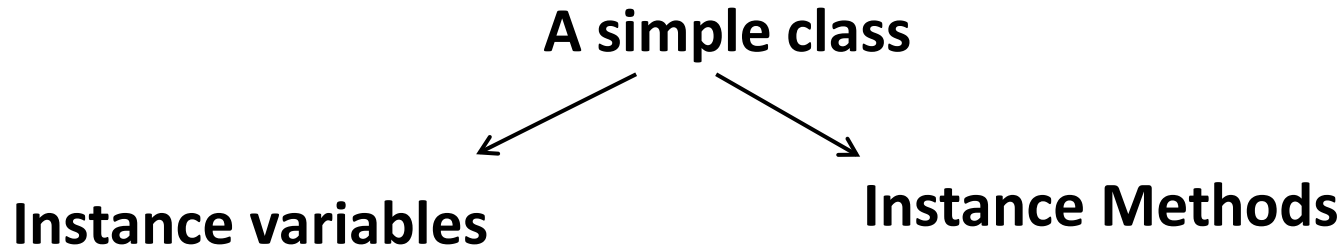
Instance vs. Static

<i>faceValue</i>	Instance variable	Every instance owns a value for each instance variable. <i>Each die has its own face value.</i>
<i>MAX_VALUE</i>	Static variable	One copy per class. Class owns its value <i>Die class owns the max face value.</i>
<i>count</i>	Static variable	One copy per class. Class owns its value <i>Die class owns the count value.</i>
<i>roll()</i>	Instance method	Behavior controlled by an instance. <i>Behavior roll should be started by each die.</i>
<i>getCount()</i>	Static method	Behavior controlled by a class. <i>Retrieves the count of dice should be started by Die class.</i>

A simple class can contain instance variables and instance methods.



A simple class contains instance variables and instance methods.



States:

The values an instance consist of.

Common behaviors:

1. Create an instance of a class (Constructor(s)).
2. Retrieve an instance variable's value(getters).
3. Change an instance variable's value to be a new value (setters).
4. Other behaviors
5. Represent an instance as a string literal.
`public String toString() {...}`
6. Compare an instance with some other object.
`public boolean equals(Object obj) {...}`

Design a class

Writing a Class, Step by Step

- A `Die` object will have the following **fields**:
 - `faceValue`
 - The `faceValue` field will hold the die's face value.
- The `Die` class will also have the following **methods**:
 - `Die` constructor(s): creates a die instance with a default face value or a specific face value referred as this die object.

Writing a Class, Step by Step

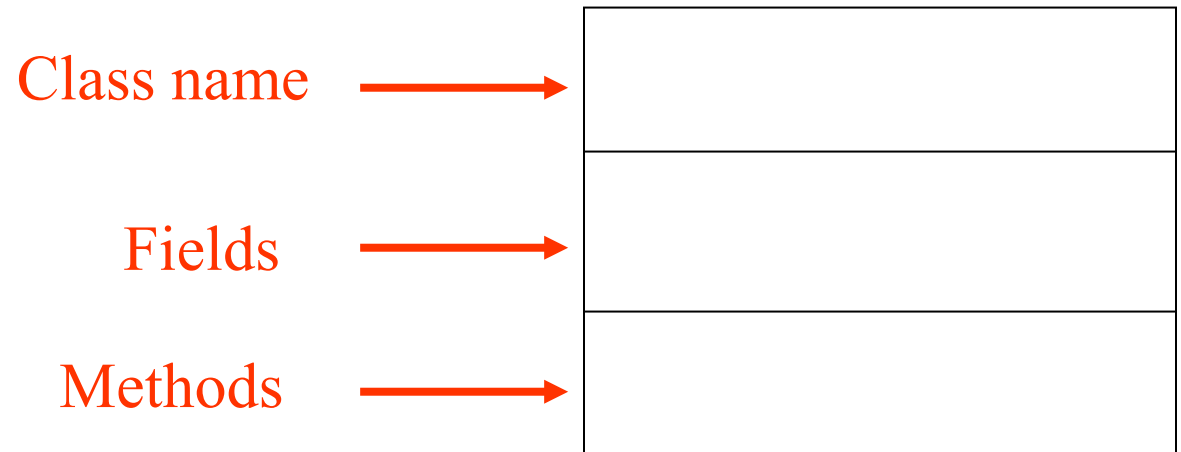
- `getFaceValue`
 - Retrieves the face value of this die.
- `setFaceValue`
 - Changes the face value of this die.
- `Roll`
 - Rolls this die.
- `equals`
 - Compares this die with some other object.
- `toString`
 - Represents this die as a string literal.

UML Diagrams

- UML stands for the *Unified Modeling Language*.
- *UML diagrams* show relationships among classes and objects.
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods).
- Lines between classes represent *associations*.
- A dotted arrow shows that one class *uses* the other (calls its methods).

UML Diagram

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting object-oriented systems.



UML Diagrams

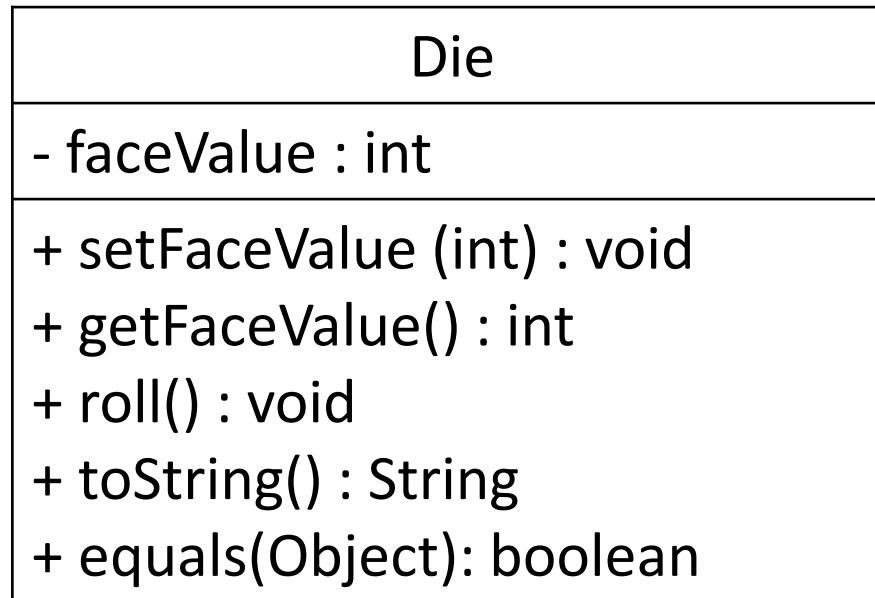
- Visibility modifier symbols:

public	+
private	-
protected	#
default	~

Access Specifiers/Visibility Modifiers

- An access specifier/visibility modifier is a Java keyword that indicates how a field or method can be accessed.
- `public`
 - When the `public` access specifier/visibility modifier is applied to a class member, the member can be accessed by code inside the class or outside.
- `private`
 - When the `private` access specifier/visibility modifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.

UML Class Diagram: Die Class



- By convention, constructors and constants should not be included in a UML class diagram.

Converting the UML Diagram to Code

Die

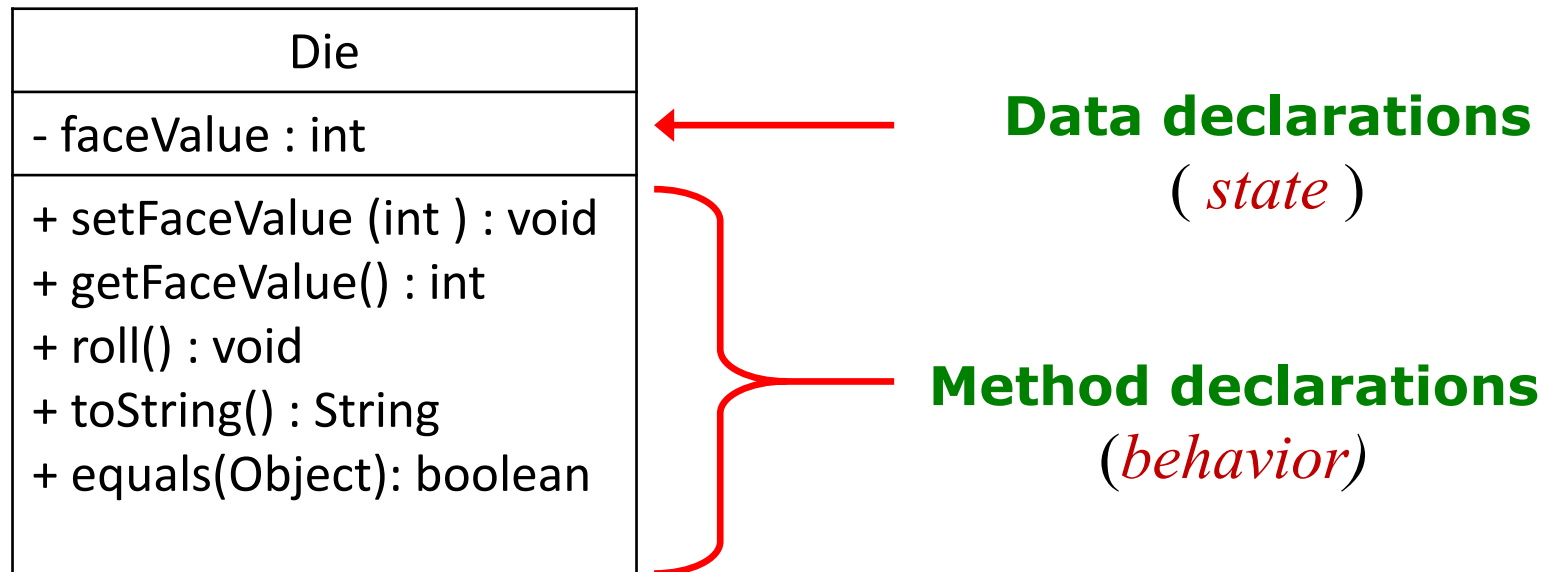
- faceValue : int

+ setFaceValue (int) : void
+ getFaceValue() : int
+ roll() : void
+ toString() : String
+ equals(Object): boolean

```
public class Die{  
    private int faceValue;  
  
    public void setFaceValue(int faceValue) {  
        ...  
    }  
    public void getFaceValue() {  
        ...  
    }  
    public void roll() {  
        ...  
    }  
    public boolean equals(Object obj) {  
        ...  
    }  
    public String toString() {  
        ...  
    }  
}
```

Classes

- A class can contain data declarations(*state*) and method declarations(*behavior*).



Classes

- The values of the data define the state of an object created from the class.
- The functionality of the methods define the behaviors of the object.

Design for the Fields

Die
- faceValue : int
+ setFaceValue (int) : void + getFaceValue() : int + roll() : void + toString() : String + equals(Object): boolean

- All instance variables should be declared as **private** variables.

Design for the Fields

Die
- faceValue : int
+ setFaceValue (int) : void + getFaceValue() : int + roll() : int + toString() : String + equals(Object): boolean

- faceValue : int



Visibility
Modifier



Type



Name



private int faceValue;

Design for the Methods

Die
- faceValue : int
+ setFaceValue (int) : void + getFaceValue() : int + roll() : void + toString() : String + equals(Object): boolean

- All instance methods should be declared as **public** methods except methods that are used as helper methods of the same class.

Design for setFaceValue

Die
- faceValue : int
+ setFaceValue (int) : void + getFaceValue() : int + roll() : void + toString() : String + equals(Object): boolean

+ setFaceValue (int) : void

Visibility Modifier Return Type Name Parameter Type Parameter Name

public void setFaceValue (int faceValue) {
...
}

Design for getFaceValue

Die
- faceValue : int
+ setFaceValue (int) : void
+ getFaceValue() : int
+ roll() : void
+ toString() : String
+ equals(Object): boolean

+ **getFaceValue() : int**

Visibility
Modifier

ReturnType

Name

public int getFaceValue() {

...

}

Design for roll

Die
- faceValue : int
+ setFaceValue (int) : void + getFaceValue() : int + roll() : void + toString() : String + equals(Object): boolean

+ roll() : void

Visibility
Modifier

ReturnType

Name

public void roll() {

...

}

Design for toString

+ toString() : String

Visibility
Modifier

ReturnType

Name

public String toString() {

...

}

Die
- faceValue : int
+ setFaceValue (int) : void + getFaceValue() : int + roll() : void + toString() : String + equals(Object): boolean

Design for equals

Die
- faceValue : int
+ setFaceValue (int) : void + getFaceValue() : int + roll() : void + toString() : String + equals(Object): boolean

+ equals(Object): boolean

Visibility
Modifier

ReturnType

Name

Parameter
Type

Parameter
Name

public boolean equals (Object obj) {

...

}

Writing Code for a class

Writing the Code for the Constructors

- A *constructor* is a special method that is used to set up an object when it is initially created.
- A constructor has the same name as the class.
- A constructor is called to create an instance.

```
/**  
 * Constructs a die with default face value of 1.  
 */  
public Die(){  
    this.faceValue = 1;  
}
```

this Reference

- The *this* reference allows an object to refer itself.
- That is, the *this* reference, used inside a method, refers to the object through which the method is being executed.
- The *this* reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same name.

Die Constructors

- Die constructor is implemented in `Die` class as follows:

```
/**  
 * Constructs a die with default face value of 1.  
 */  
public Die(){  
    this.faceValue = 1;  
}
```

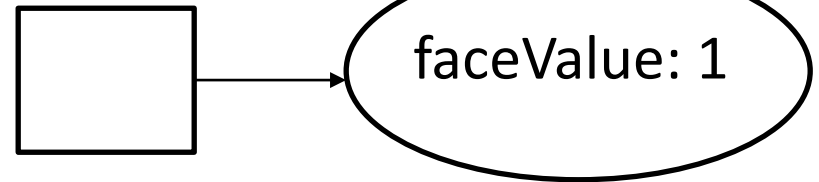
Driver Programs

- A *driver program* drives the use of other, more interesting parts of a program.
- Driver programs are often used to test other parts of the software.
- A driver programs contains a `main` method that drives the use of the `Die` class, exercising its services.

Die Constructors

```
public Die(){  
    this.faceValue = 1;  
}
```

firstDie




- In other methods such as `main`, Die instances can be created as follows:

```
public static void main(String[] args){  
    int firstDieFaceValue;  
    String str = new String("Hello");  
    → Die firstDie = new Die();  
    Die secondDie = new Die(5);  
}
```

Die Constructors

- Die constructor can be implemented in different version as follows:

```
/**
 * Constructs a die with a face value.
 * @param faceValue The face value of this die
 */
public Die(int faceValue){
    if(faceValue >= 1 && faceValue <= 6){
        this.faceValue = faceValue;
    }
}
```

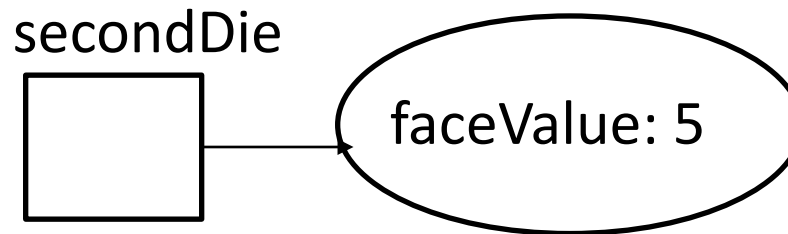


this.faceValue: instance variable
faceValue: local parameter

- By convention, a method parameter should name after the corresponding instance variable.
- For example, parameter *faceValue* is named after instance variable *faceValue*.

Using Die Constructors

```
public Die(int faceValue){  
    if(faceValue >= 1 && faceValue <= 6){  
        this.faceValue = faceValue;  
    }  
}
```



- In a driver program, Die instances can be created as follows:

```
public static void main(String[] args){  
    int firstDieFaceValue;  
    String str = new String("Hello");  
    Die firstDie = new Die();  
    → Die secondDie = new Die(5);  
}
```

The Default Constructor

- When an object is created, its constructor is always called.
- If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*.
 - It sets all of the object's numeric fields to 0.
 - It sets all of the object's `boolean` fields to `false`.
 - It sets all of the object's reference variables to the special value *null*.

The Default Constructor

- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The only time that Java provides a default constructor is when you do not write any constructor for a class.
- A default constructor is not provided by Java if a constructor is already written.

Writing Your Own No-Arg Constructor

- A constructor that does not accept arguments is known as a *no-arg constructor*.
- The default constructor (provided by Java) is a no-arg constructor.
- We can write our own no-arg constructor

```
/**
 * Constructs a die with default face value of 1.
 */
public Die(){
    this.faceValue = 1;
}
```

Constructors

- Note that a constructor has no return type specified in the method header, not even `void`.
- A common error is to put a return type on a constructor, which makes it a “regular” method that happens to have the same name as the class.

Writing code for getFaceValue

- There are also methods that explicitly retrieve the current face value at any time.
- It is called getter/accessor.
- A getter should be a public method.

```
/**
 * Returns the face value of this die.
 * @return An integer specifying the face value of this die
 */
public int getFaceValue(){
    return this.faceValue;
}
```

Calling getFaceValue

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

- Retrieve the face value of first die.

```
public static void main(String[] args){  
    int firstDieFaceValue;  
    String str = new String("Hello");  
    Die firstDie = new Die();  
    Die secondDie = new Die(5);
```

firstDie



faceValue: 1

```
//Retrieves the face value of first die.  
firstDieFaceValue = firstDie.getFaceValue();
```



A copy of 1 is
returned.

Writing code for setFaceValue

- There are also methods that explicitly change/ set the current face value at any time.
- It is called setter/mutator.
- A setter should be a public method.

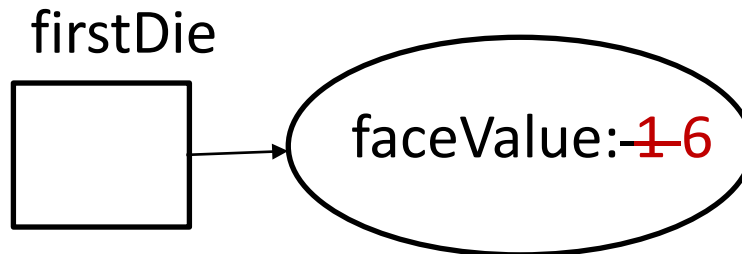
```
/**
 * Changes this die's face value to be a new value
 * that is between 1 and 6.
 * @param faceValue An integer specifying a new face value of this die
 */
public void setFaceValue(int faceValue){
    if(faceValue >= 1 && faceValue <=6){
        this.faceValue = faceValue;
    }
}
```

Calling setFaceValue

```
public void setFaceValue(int faceValue){  
    if(faceValue >= 1 && faceValue <=6){  
        this.faceValue = faceValue;  
    }  
}
```

- Change the face value of first die.

```
//Changes the face value of first die.  
→ firstDie.setFaceValue(6);  
firstDieFaceValue = firstDie.getFaceValue();  
System.out.println(firstDieFaceValue); //6
```



1 is changed to 6.

Writing code for roll

- The `roll` method changes this die's face value to be a random value between 1 and 6.
- This method should be a public method.

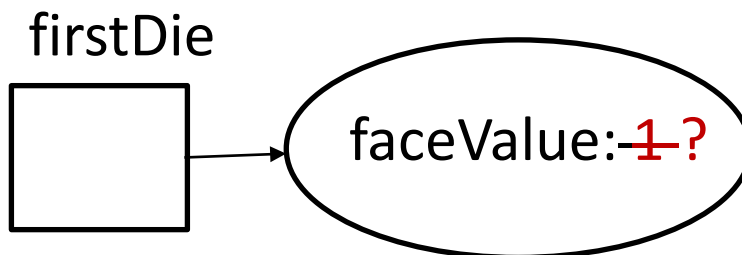
```
/**
 * Changes this die's face value to be a random value between 1 and 6.
 */
public void roll(){
    //Randomly select a value between 1 and 6
    Random generator = new Random();
    int randomFaceValue = generator.nextInt(6) + 1;
    //Changes this die's face value
    this.faceValue = randomFaceValue;
}
```


Calling roll

```
public void roll(){
    //Randomly select a value between 1 and 6
    Random generator = new Random();
    int randomFaceValue = generator.nextInt(6) + 1;
    //Changes this die's face value
    this.faceValue = randomFaceValue;
}
```

- Changes first die's face value randomly.

```
//Roll the first die
firstDie.roll();
firstDieFaceValue = firstDie.getFaceValue();
System.out.println(firstDieFaceValue);
```



1 is changed randomly.
?: 1-6

Writing code for toString

- All classes that represent objects should define a `toString` method that overrides the `toString` method in the ultimate super class `Object`.
- The `toString` method returns a character string that represents the object in some way.

```
/**
 * Returns a string representation of this die containing the
 * type and hidden values.
 * @return A string representation of this die
 */
public String toString(){
    //the type and hidden values
    return this.getClass().getSimpleName() + ": The face value is " + this.faceValue;
}
```

Calling toString

- It is called automatically when an object is concatenated to a string or when it is passed to the `println` method.

```
System.out.println(firstDieFaceValue);
```

Writing code for equals

- All classes that represent objects should define a `equals` method that overrides the `equals` method in the ultimate super class `Object`
- The `equals` method compares instance data, returns true if the values of this object's instance data are the same as the values of the other object's instance data

Writing code for equals

```
/**
 * Indicates if this die is "equal to" some other object. If the other object is not a die,
 * return false. If the other object is a die, and has the same face value, return true.
 * @param obj An Object reference to a specific object
 * @return A boolean value specifying whether this die is equal to some other object
 */
public boolean equals(Object obj){
    //check the type of the specific object.
    //if it is not a die, return false.
    if(!(obj instanceof Die)){
        return false;
    }

    //If it is a die, compare the values of this die with
    //the values of the specific die, return the comparison result.
    Die other = (Die)obj;

    return this.faceValue == other.faceValue;
}
```

Calling equals

```
//Compares the first die with other objects.  
System.out.println(firstDie.equals(secondDie));  
System.out.println(firstDie.equals(str));  
System.out.println(firstDie.equals(firstDie));
```

False if the first die and second die have difference face value.
True if the first die and second die have same face value.

False since first die isn't equal to a string.

True since first die is equal to itself.

Data Scope

Data Scope

- The *scope* of data is the area in a program in which that data can be referenced (used).
- Data declared at the class level can be referenced by all methods in that class.
- Data declared within a method can be used only in that method.
- Data declared within a method is called *local data* (*local variable*).

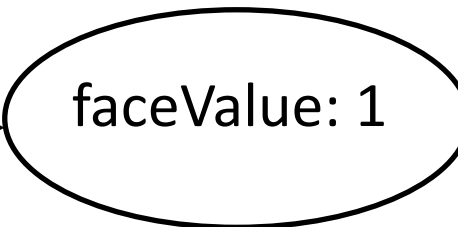
Instance Data(Instance Variables)

- The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) that is created has its own version of it.
- A class declares the type of the data, but it does not reserve any memory space for it.

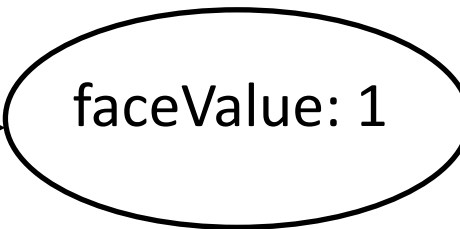
Instance Data(Instance Variables)

- Every time a `Die` object is created, a new `faceValue` variable is created as well.

firstDie



secondDie



Instance Data(Instance Variables)

- The objects of a class share the method definitions, but each object has its own data space.
- That's the only way two objects can have different states.
- Each object maintains its own faceValue variable, and thus its own state.

Instance Data vs. Local Data

- The number of instance variables is how many values each instance can store.
- A local variable is a temporary space to store a temporary value in a method.

Instance Data vs. Local Data

- See Die class and Die2 class.

```
public class Die{  
    private int faceValue;  
    ...
```

Good design:

In Die class, a die stores one value – a face value.

faceValue: 1

Instance Data vs. Local Data

- See Die class and Die2 class.

Bad design:

In Die2 class, a die stores two values – a face value and a random face value. Variable `randomFaceValue` should be a local variable in `roll` method.

```
public class Die2{  
    private int faceValue;  
    int randomFaceValue;  
    ...  
  
    public void roll(){  
        Random generator = new Random();  
        int randomFaceValue;  
        ...  
    }  
}
```



faceValue: 1
randomFaceValue: 4

Encapsulation

Encapsulation

- We can take one of two views of an object:
 - internal - the details of the variables and methods of the class that defines it
 - external - the services that an object provides and how the object interacts with the rest of the system
- From the external view, an object is an *encapsulated* entity, providing a set of specific services.
- These services define the *interface* to the object.

Encapsulation

- One object (called the *client*, such as the `Driver` program) may use another object for the services it provides(such as `roll` method).
- The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished.
- Any changes to the object's state (its variables) should be made by that object's methods.
- That is, an object should be *self-governing*.

Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*.
- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data.
- Java has three visibility modifiers: `public`, `protected`, and `private`.
- The `protected` modifier involves inheritance.

Visibility Modifiers

- Members of a class that are declared with *public visibility* can be referenced anywhere.
- Members of a class that are declared with *private visibility* can be referenced only within that class.
- Members declared without a visibility modifier have *default visibility* and can be referenced by any class in the same package.

Visibility Modifiers

- Public variables violate encapsulation because they allow the client to “reach in” and modify the values directly.
- Therefore instance variables should not be declared with public visibility.
- It is acceptable to give a constant public visibility, which allows it to be used outside of the class.
- Public constants do not violate encapsulation because, although the client can access it, its value cannot be changed.

Visibility Modifiers

- Methods that provide the object's services are declared with public visibility so that they can be invoked by clients.
- Public methods are also called *service methods*.
- A method created simply to assist a service method is called a *support method*.
- Since a support method is not intended to be called by a client, it should not be declared with public visibility.

Visibility Modifiers

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Accessors and Mutators

- Because instance data is private, a class usually provides services to access and modify data values.
- An *accessor*(getter) *method* returns the current value of a variable.
- A *mutator* (setter)*method* changes the value of a variable.
- The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where X is the name of the value.

Mutator Restrictions

- The use of mutators gives the class designer the ability to restrict a client's options to modify an object's state.
- A mutator is often designed so that the values of variables can be set only within particular limits.
- For example, the `setFaceValue` mutator of the `Die` class should have restricted the value to the valid range (1 to `MAX`).

Local Data

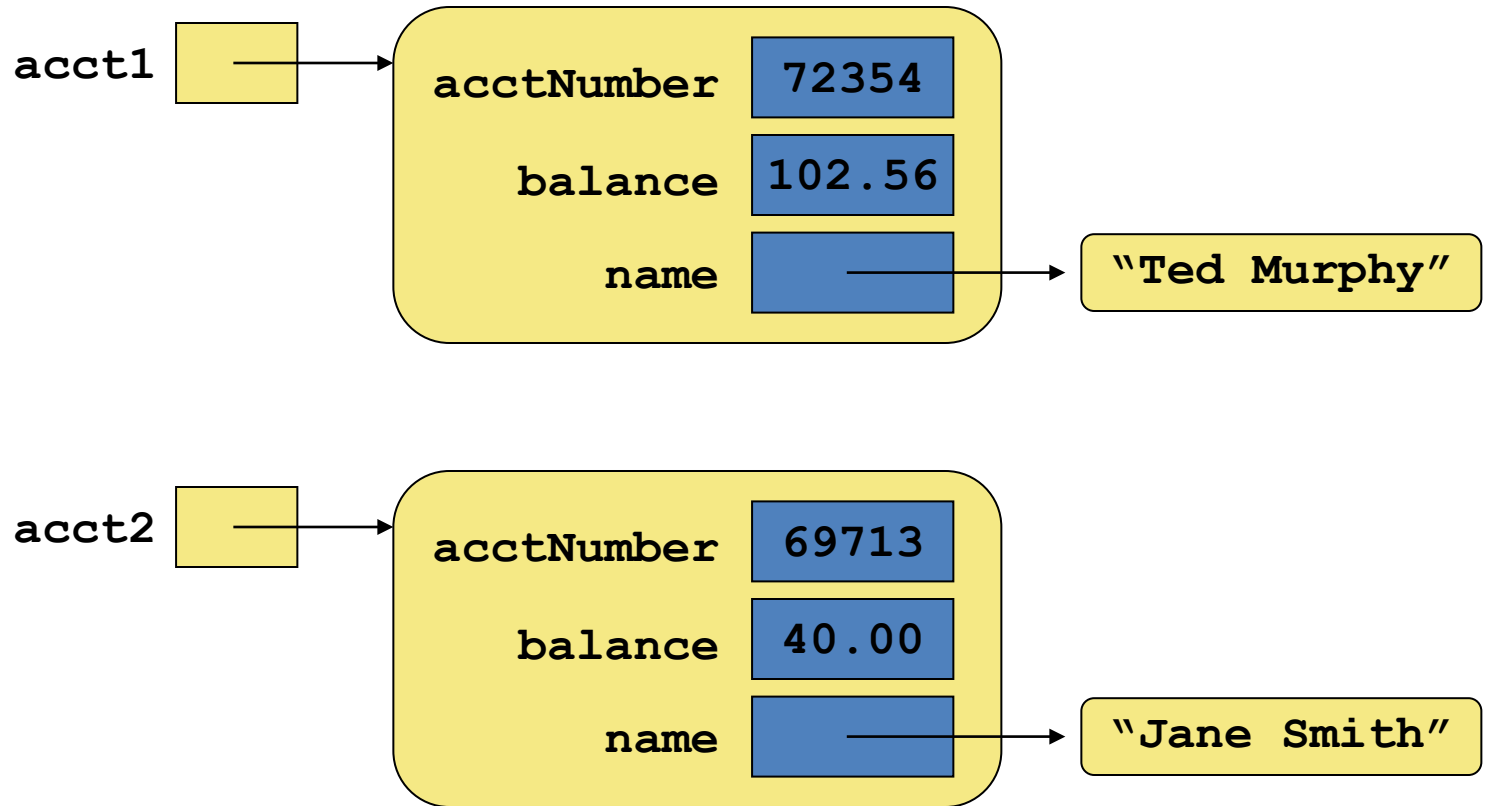
- As we've seen, local variables can be declared inside a method.
- The formal parameters of a method create *automatic local variables* when the method is invoked.
- When the method finishes, all local variables are destroyed (including the formal parameters).
- Keep in mind that instance variables, declared at the class level, exists as long as the object exists.

Bank Account Example

- Let's represent a bank account by a class named `Account`.
- States:
 - the account number
 - the current balance,
 - the name of the owner
- Behaviors (or services):
 - deposits
 - withdrawals
 - adding interest

Account
<ul style="list-style-type: none">- acctNumber: long- balance :double- name: String
<ul style="list-style-type: none">+ deposit (double): void+ withdraw (double, double): double+ addInterest (): double+ getBalance (): double+ getName(): String+ setName(String): void+ getAcctNumber(): long+ toString(): String+ equals(Object): boolean

Bank Account Example



Overloading Methods and Constructors

Overloading Methods and Constructors

- Two or more methods in a class may have the same name as long as their parameter lists are different.
- When this occurs, it is called *method overloading*. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.

Overloaded Method add

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

```
public String add (String str1, String str2) {  
    String combined = str1 + str2;  
    return combined;  
}
```

Method Signature and Binding

- A method signature consists of the method's name and the data types of the method's parameters, in the order that they appear. The return type is not part of the signature.

`add(int, int)`

`add(String, String)`

Signatures of the add
methods of previous
slide

- The process of matching a method call with the correct method is known as *binding*. The compiler uses the method signature to determine which version of the overloaded method to bind the call to.

Die Class Constructor Overload

```
/**
 * Constructs a die with default face value of 1.
 */
public Die(){
    this.faceValue = 1;
}
/**
 * Constructs a die with a face value.
 * @param faceValue The face value of this die
 */
public Die(int faceValue){
    if(faceValue >= 1 && faceValue <= 6){
        this.faceValue = faceValue;
    }
}

    public static void main(String[] args){
        int firstDieFaceValue;
        String str = new String("Hello");
        Die firstDie = new Die();
        Die secondDie = new Die(5);
    }
}
```


Static Class Members

- *Static fields* and *static methods* do not belong to a single instance of a class.
- To invoke a static method or use a static field, the class name, rather than the instance name, is used.
- Example:

```
double val = Math.sqrt(25.0);
```



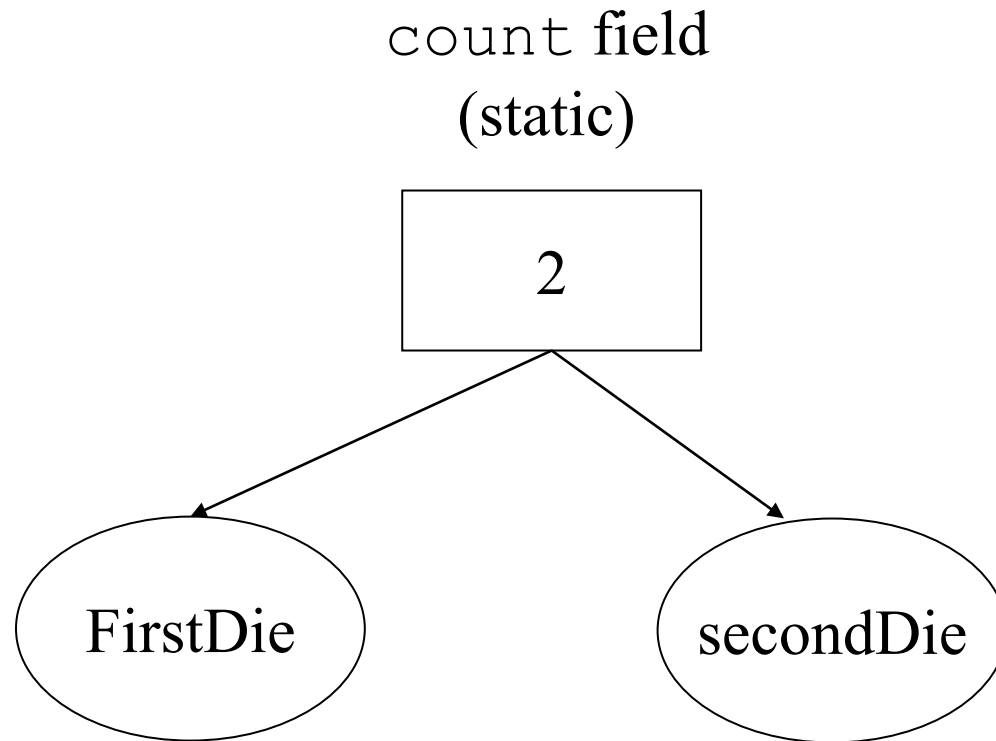
Class name

Static method

Static Fields

- Class fields are declared using the `static` keyword between the access specifier and the field type.
`private static int count = 0;`
- The field is initialized to 0 only once, regardless of the number of times the class is instantiated.
 - Primitive static fields are initialized to 0 if no initialization is performed.

Static Fields



Static Methods

- Methods can also be declared static by placing the `static` keyword between the access modifier and the return type of the method.

```
/**
 * Returns the count of die objects.
 * @return The count of die object
 */
public static int getCount(){
    return count;
}
```

- When a class contains a static method, it is not necessary to create an instance of the class in order to use the method.

Static Methods

- Static methods are convenient because they may be called at the class level.
- They are typically used to create utility classes, such as the `Math` class in the Java Standard Library.
- Static methods may not communicate with instance fields, only static fields.

Summary

- Objects and Classes
- Anatomy of a Class
 - Class components
 - Design a class
 - Write code for a class
- Data Scope
- Encapsulation
- Overloading Methods and Constructors
- Static members of a class