We have used a number of Java classes: Scanner, String, Random, Math, Character ....

Now we consider defining our own classes

A couple of quick examples:

– PlayingCard
– Word
public class PlayingCard {
    private String suit;
    private String face;

    public PlayingCard(String s, String f) {
        suit = s;
        face = f;
    }

    public String toString() {
        return face + " of " + suit;
    }
}

public class UsePlayingCards {
    public static void main(String[] args) {
        PlayingCard p1 = new PlayingCard(...);
        System.out.println(p1);
    }
}
public class Word {
    private String text;
    private int frequency;

    public Word(String w) {
        text = w;
        frequency = 1;
    }

    public String toString() {
        return text;
    }
}

public class ProcessWords {
    public static void main(String[] args) {
        Word w = new Word("Java");
        System.out.println(w);
    }
}
Classes comprise fields and methods

Fields:
- Things that describe the class or describe instances (i.e. objects)
- e.g. student number, first name, last name, gender, ...

Methods:
- constructors, getters, setters, other...
- e.g. getFirstName(), setFirstName(), equals()

A getter/accessor
A setter/mutator
<table>
<thead>
<tr>
<th>Class name</th>
<th>A class has a name,</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fields</strong></td>
<td>Fields (called variables in Java code)</td>
</tr>
<tr>
<td><strong>methods</strong> (including constructors)</td>
<td>Methods (A method may have local fields)</td>
</tr>
</tbody>
</table>
A quick look at two classes we have used: Math and Random

**Math** provides some useful utility methods. We use it without instantiating an object.

```java
double area = Math.PI * Math.pow(r, 2);
```

**Random** lets us use random sequences. To utilize this we must instantiate objects.

```java
Random die = new Random();
int toss = die.nextInt(6) + 1;
```

<table>
<thead>
<tr>
<th>Math</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>+E</td>
<td>-seed</td>
</tr>
<tr>
<td>+PI</td>
<td>-multiplier</td>
</tr>
<tr>
<td>-Math()</td>
<td>+Random()</td>
</tr>
<tr>
<td>+abs(double a)</td>
<td>+Random(long seed)</td>
</tr>
<tr>
<td>+abs(float a)</td>
<td>+nextBoolean()</td>
</tr>
<tr>
<td>+ abs(int a)</td>
<td>+nextInt()</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>+ max(double a, double b)</td>
<td>+nextInt()</td>
</tr>
<tr>
<td>+ max(int a, int b)</td>
<td>...</td>
</tr>
</tbody>
</table>
Math has two **static** fields

Math has a **private** constructor

You cannot instantiate a Math object

Math has many **static** methods

To use $\pi$ you write

Math.PI

To use the static method `max` you write

Math.max(n1, n2)

---

**Here we specify the name of the class**

<table>
<thead>
<tr>
<th>Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>+E</td>
</tr>
<tr>
<td>+PI</td>
</tr>
<tr>
<td>-Math()</td>
</tr>
<tr>
<td>+abs(double a)</td>
</tr>
<tr>
<td>+abs(float a)</td>
</tr>
<tr>
<td>+ abs(int a)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>+ max(double a, double b)</td>
</tr>
<tr>
<td>+ max(int a, int b)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
We must instantiate an object to get a random sequence

```java
Random gen = new Random();
```

Random has *some private* instance fields

- `seed`
- `multiplier = 0x5DEECE66DL`

**Constructors**

- `Random()`
- `Random(long seed)`

**Instance methods**

- `gen.nextBoolean()`
- `gen.nextInt()`

... an instance
Fields

Fields may be primitive variables
Or, they may be of some other type
e.g. String, PlayingCard, Word

May be public or private
  public  – anyone can use it
  private  – limited access
• Methods are either:
  – value-returning
    must have a return statement
    e.g. getters \(\leftarrow\) naming convention is ...
  – void
    no return statement
    e.g. setters \(\leftarrow\) naming convention is ...
• public vs private
  public - anyone can use it
  private - special cases

  Math constructor is private – you cannot instantiate a Math object ... try to do it
• All classes should have

   equals(...) 

   toString()
equals(...)

- Value-returning
- Returns a boolean
- Usually an equals method is designed for a class. Designer must determine the condition when two objects are considered equal.
- E.g. String class has an equals method
  ```java
  string1.equals(string2)
  "abc".equals("xyz")  returns false
  "abc".equals("abc")  returns true
  ```
**toString Method**

**toString()**

- Value-returning
- Returns a string
- A method automatically called when an object is displayed
  
  E.g. `System.out.println(myObject);`
- The designer of a class determines what it returns
- E.g. ArrayList has a `toString()` method ... result is of the form:
  
  `[ object₁, object₂, ... objectₙ ]`
Consider the student class in the text →
# Class Diagram for Student

## Fields
- id
- firstName
- lastName
- gender
- active

## Constructors
- `+Student ()`
- `+Student (firstName, lastName, gender, active)`
- `-nextId ()`
- `+getId ()`
- `+getLastId ()`
- `+getFirstName ()`
- `+getLastName ()`
- `+getGender ()`
- `+isActive ()`
- `+getMajor ()`
- `+setLastId (newLastId)`
- `+setFirstName (newFirstName)`
- `+setLastName (newLastName)`
- `+setGender (newGender)`
- `+setActive (newActive)`
- `+setMajor (newMajor)`
- `+toString ()`
- `+equals (s)`

## Show 1, 2, or 3 compartments / info as needed

* + means there is public access to the method
* - means there is no public access to the field or method
instance vs class

e.g. consider Student class

Which fields are class?

Which fields are instance?
instance vs class

*Instance* \(\equiv\) an *object*

*Static field* \(\equiv\) *class-level field*

Regardless of the number of students there is only one *lastId* field. It is a class-level field that is shared by all Student instances.

There are *id*, *firstName*, *lastName*, *gender*, *active*, and *major* fields for each Student *instance*. So each student can have different values.
private vs public

private:
only directly accessible from within the class/object,
and from outside the class via getters/setters

public: accessible from anywhere

A design principle is to make **fields private**
but give **public access to the getters** and setters (a later slide)
Java code for Student - constructors

The no-arg constructor

```java
public Student() {
    id = nextId();
    // default values for a student:
    firstName = "unknown";
    lastName = "unknown";
    gender = '?';
    active = false;
}

// second constructor, four arguments
public Student (String firstName, String lastName, char gender, boolean active) {
    id = nextId();
    //
    // when parameters and fields have the same
    // name they are distinguished this way:
    // a field name alone refers to the
    // a field name prefixed with "this."
    // refers to an object’s fields.
    this.firstName = firstName;
    this.lastName = lastName;
    this.gender = gender;
    this.active = active;
}
```

Use as many constructors as your application requires.
Constructors differ in the number and type of parameters.
Notice

Getters (also called accessors) for most private fields

```java
public String getFirstName(){
    return firstName;
}
public String getLastName(){
    return lastName;
}
public char getGender(){
    return gender;
}
public boolean isActive(){
    return active;
}
```

**Naming convention**:
- Start with “get” followed by the field name but this starts with a capital letter.
- **Naming convention for boolean**: Start with “is” followed by the field name but this starts with a capital letter.
Notice

Setters (also called mutators) for most private fields

```java
public void setFirstName(String newFirstName){
    firstName = newFirstName;
}
public void setLastName(String newLastName){
    lastName = newLastName;
}
public void setGender(char newGender){
    gender = newGender;
}
public void setActive(boolean newActive){
    active = newActive;
}
```

**Naming convention:**
Start with “set” followed by the field name but this starts with a capital letter.
Java code for Student – other methods

```
private int nextId(){
    // increment lastId and return the new value
    // to be used for the new student.
    return ++lastId;
}
```

private method `nextId`
Used to control the id assigned to a new student object

```
public String toString(){
    return id+ " "+firstName+ " "+lastName;
}
```

`toString`
Executes when a student is printed

```
public boolean equals(Student s){
    return id == s.id;
}
```

equals
Tests two student objects to see if they are ‘equal’
Class is a template for objects

How are these shown in UML?

UML = unified modeling language
Objects:

- instantiated/created via `new` – lots of examples
- also called an *instance* – so we can speak of instance fields/methods

How are these shown in UML?

---

**Figure 7.3: Object Diagram with 3 student objects.**

- Underlined object name followed by “:” followed by class name
- Field values
Listing 7.1:

Creates two students
One using the no-arg constructors and setters
The other using a 4-arg constructor
/**
 * Create two student objects
 * using the two constructors
 */

public class UseConstructors {

    public static void main (String[] args){
        // first, with the no-arg constructor
        Student jill = new Student();
        // use setters to complete the student object
        jill.setFirstName("Jill");
        jill.setLastName("Lee");
        jill.setGender('F');
        jill.setActive(true);
        // now with the other constructor
        Student sam = new Student("Samantha","Jones","F",true);
        // display the students
        System.out.println(jill);
        System.out.println(sam);
    }
}