

Classes and Objects: A Deeper Look

CS102A Lecture 9

James YU

Nov. 9, 2020



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

static vs. instance variables

- Recall that every object of a class has its own copy of all the instance variables of the class.
 - Instance variables represent concepts that are unique per instance, e.g., `name` in class `Student`.
- In certain cases, only one copy of a particular variable should be shared by all objects of a class (e.g., a counter that keeps track of every object created for memory management).
 - A `static` field, called a *class variable* - is used in such cases.



When to use `static` field

- Suppose we have a video game with Soldiers fighting Monsters.
- Each Soldier tends to be brave and is willing to attack if there is at least four other Soldiers on the field.
- Each Soldier object needs to know the `soldierCount`.
- Should we make the `soldierCount` a static variable or instance variable?
- If we make this field an instance variable, we need to update this field in every `Soldier` objects.
- So we should make this field a `static` variable.



Recall your lab exercise

- In your lab, each Circle object has separate `x`, `y` and `radius`.
- What if we add a `static` variable controlling the thickness of all the circles?

static class members

- A **static** variable represents *class-wide information*. All objects of the class share the same piece of data.

```
1 public class Employee {  
2     private String firstName;  
3     private String lastName;  
4     private static int count;  
5 }
```

- There will be a new copy whenever a new object is created.
- There is only one copy for each **static** variable. Make a variable static when all objects of the class must use the same copy of the variable.



static class members

- `static` class members are available as soon as the class is loaded into memory at execution time (objects may not exist yet).
- A class's `public static` members can be accessed through a reference to any object of the class, or by qualifying the member name with the *class name* and a dot (`.`), e.g., `Math.PI`.

```
1 public class Employee { ...
2     public static int count; // number of employees created
3     public static void main(String[] args) {
4         Employee e = new Employee();
5         System.out.printf("# employees = %d", e.count); // not encouraged
6         System.out.printf("# employees = %d", Employee.count); // good
           practice
7     }
8 }
```



static class members

- A class's **private static** members can be accessed by client code *only through methods* of the class.

```
1 public class Employee {  
2     private String firstName;  
3     private String lastName;  
4     private static int count; // number of employees created  
5     public static int getCount() { return count; }  
6     public static void main(String[] args) {  
7         System.out.printf("# employees = %d", Employee.getCount());  
8     }  
9 }
```



static class members

- A `static` method cannot access non-static class members (e.g., instance variables), because a static method can be called even when no objects of the class have been instantiated.
- An instance method can access `static` class members, but not other way round.
- If a static variable is not initialized, the compiler assigns it a default value (e.g., `0` for `int`).

Example

```
1 public class Employee {  
2     private String firstName;  
3     private String lastName;  
4     private static int count; // number of employees created  
5     public Employee(String first, String last) {  
6         firstName = first;  
7         lastName = last;  
8         ++count;  
9         System.out.printf("Employee constructor: %s %s; count = %d\n",  
10             firstName, lastName, count);  
11     }  
12     public String getFirstName() { return firstName; }  
13     public String getLastName() { return lastName; }  
14     public static int getCount() { return count; }  
15 }
```



Example

```
1 public class EmployeeTest {
2     public static void main(String[] args) {
3         System.out.printf("Employees before instantiation: %d\n",
4             Employee.getCount());
5         Employee e1 = new Employee("Bob", "Blue");
6         Employee e2 = new Employee("Susan", "Baker");
7         System.out.println("\nEmployees after instantiation:");
8         System.out.printf("via e1.getCount(): %d\n", e1.getCount());
9         System.out.printf("via e2.getCount(): %d\n", e2.getCount());
10        System.out.printf("via Employee.getCount(): %d\n", Employee.getCount
11            ());
12        System.out.printf("\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
13            e1.getFirstName(), e1.getLastName(),
14            e2.getFirstName(), e2.getLastName());
15    }
}
```



Example

```
Employees before instantiation: 0  
Employee constructor: Bob Blue; count = 1  
Employee constructor: Susan Baker; count = 2
```

```
Employees after instantiation:  
via e1.getCount(): 2  
via e2.getCount(): 2  
via Employee.getCount(): 2
```

```
Employee 1: Bob Blue  
Employee 2: Susan Baker
```



static import

- Normal import declarations import classes from packages, allowing them to be used without package qualification
 - `import java.util.Scanner;`
 - Otherwise `java.util.Scanner input = new java.util.Scanner(System.in);`
- A static import declaration enables you to import the static members of a class so you can access them via their unqualified names, i.e., without including class name and a dot (`.`)
 - `Math.sqrt(4.0) → sqrt(4.0)`



static import

- Import a particular static member (single static import)
 - `import static packageName.ClassName.staticMemberName;`
 - The member could be a field or a method.
- Import all static members of a class (static import on demand)
 - `import static packageName.ClassName.*;`
 - * is a wildcard, meaning “matching all”.



Example

```
1 // Static import of Math class methods.
2 import static java.lang.Math.*;
3 public class StaticImportTest
4 {
5     public static void main(String[] args)
6     {
7         System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
8         System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
9         System.out.printf("E = %f%n", E);
10        System.out.printf("PI = %f%n", PI);
11    }
12 } // end class StaticImportTest
```

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```



Notes on set and get methods

- Recall your lab exercise of designing the class `Circle` again.
- There are three private instance variables:

```
1 private double x;  
2 private double y;  
3 private double radius;
```

- There is a pair of set and get methods for each instance variable:
 - `getRadius()`, `setRadius(double)`, `getX()`, `setX(double)`, `getY()`, `setY(double)`.



Notes on set and get methods

- Classes often provide **public** methods to allow clients to set (i.e., assign values to) or get (i.e., obtain the values of) **private** instance variables.
- **Set methods** are also called *mutator methods*, because they typically change an object's state by modifying the values of instance variables.
- **Get methods** are also called **accessor methods** or **query methods**.

```
1 private int hour;  
2 public void setHour(int h) { hour = ( ( h >= 0 && h < 24 ) ? h : 0 );  
    }  
3 public int getHour() { return hour; }
```



Notes on set and get methods

- Why not making the fields `public`?

```
1 public class Date {  
2     private int year;  
3     private int month;  
4     private int day;  
5     public int getYear() {  
6         return year;  
7     }  
8     public void setYear(int y) {  
9         year = y;  
10    }  
11    public int getMonth() {  
12        . . . . .  
13    }
```

```
14    public void setMonth(int m) {  
15        . . . . .  
16    }  
17    public int getDay() {  
18        . . . . .  
19    }  
20    public void setDay(int d) {  
21        . . . . .  
22    }  
23 }
```



Notes on set and get methods

```
1 public class DateTest {  
2     public static void main() {  
3         Date date = new Date();  
4         date.setMonth(10);  
5         System.out.println(date.getMonth());  
6     }  
7 }
```

```
1 public class DateTest {  
2     public static void main() {  
3         Date date = new Date();  
4         date.month = 10;  
5         System.out.println(date.month);  
6     }  
7 }
```

```
1 public class Date {  
2     public int year;  
3     public int month;  
4     public int day;  
5 }
```



Restrict users from setting invalid values

```
1 public class DateTest {  
2     public static void main() {  
3         Date date = new Date();  
4         date.month = 13;  
5         date.day = 40;  
6         System.out.println(date.month);  
7         System.out.println(date.day);  
8     }  
9 }
```

```
1 public class Date {  
2     public int year;  
3     public int month;  
4     public int day;  
5 }
```



Do checking in the set methods

```
1 public class DateTest {  
2     public static void main() {  
3         Date date = new Date();  
4         date.setMonth(13);  
5         System.out.println(date.  
6             getMonth());  
7     }  
}
```

```
1 public class Date {  
2     private int year;  
3     private int month;  
4     private int day;  
5     . . . . .  
6     public int getMonth() {  
7         . . . . .  
8     }  
9     public void setMonth(int m) {  
10         if ( m >= 1 && m <= 12)  
11             month = m;  
12     }  
13     . . . . .  
14 }
```



It breaks *encapsulation*

- The core principle here is that the fields of your class are implementation details, by exposing it as a `public` field, you are telling everything outside that class that it is an actual piece of data that is stored by the class – something external classes don't need to know, they just need to be able to get or set that piece of data.
- As soon as you make something `public`, external classes should be able to depend on it, and it should not change often, by implementing as a method, you are retaining the flexibility to change the implementation at a later point without affecting users of the class.

Encapsulation in Java

- **Encapsulation** is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Other way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.
 - Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
 - As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.
 - Encapsulation can be achieved by: Declaring all the variables in the class as **private** and writing **public** methods in the class to set and get the values of variables.



Key technologies of OOP

- Some people's comment on the internet:
 - *Encapsulation* is one of the four fundamental OOP concepts. The other three are *inheritance*, *polymorphism*, and *abstraction*.
- In the text book:
 - *Classes*, *objects*, *encapsulation*, *inheritance* and *polymorphism* – the key technologies of object-oriented programming.
- Java is designed to solve problems, not designed to test your intelligence.



More on data hiding and integrity

- It seems that providing set and get capabilities is essentially the same as making the instance variables **public**.
 - A **public** instance variable can be read or written by any method that has a reference to an object that contains that variable.
 - If an instance variable is declared **private**, a **public** get method certainly allows other methods to access it, but the get method can control how the client can access it.
 - A **public** set method can – and should – carefully validate attempts to modify the variable's value to ensure that the new value is valid for that data item.



final instance variables

- The *principle of least privilege* is fundamental to good software engineering.
 - Code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more.
 - Makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.



final instance variables

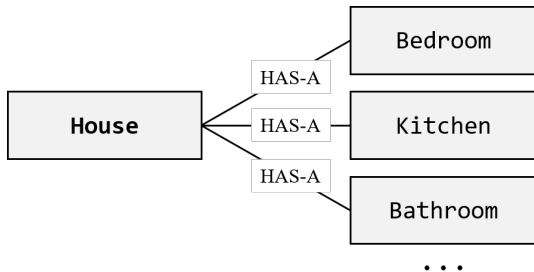
- The keyword `final` specifies that a variable is not modifiable (i.e., constant) and any attempt to modify leads to an error (cannot compile).

```
1 private final int INCREMENT;
```

- `final` variables can be initialized when they are declared.
- If they are not, they must be initialized in every constructor of the class.
 - Initializing `final` variables in constructors enables each object of the class to have a different value for the constant.
- If a `final` variable is not initialized when it is declared or in every constructor, the program will not compile.

Composition

- A class can have references to objects of other classes as members.
- This is called composition and is sometimes referred to as a *has-a* relationship.











Designing an `Employee` class

- Suppose we are designing an employee management system, what information should be included in the `Employee` class?
 - First name (`String` type);
 - Last name (`String` type);
 - Date of birth (? type);
 - Date of hiring (? type);
 - ... potentially lots of other information.



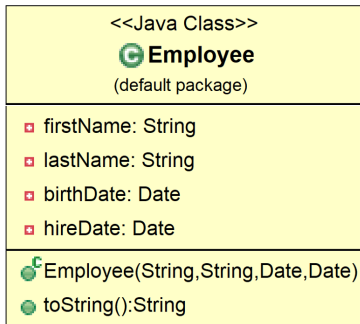
Let's define a **Date** class

- What kind of information (stored in instance variables) should be included?
- What kind of operations (methods) should be included?

<<Java Class>>	
 Date (default package)	
	month: int
	day: int
	year: int
	Date(int,int,int)
	checkMonth(int):int
	checkDay(int):int
	toString():String



Define the **Employee** class



Let's look at the real code

```
1 public class Date {  
2     private int year;  
3     private int month;  
4     private int day;  
5 }
```

<<Java Class>>


 **Date**

(default package)

▣ month: int


▣ day: int

▣ year: int

 Date(int,int,int)

▣ checkMonth(int):int

▣ checkDay(int):int

 toString():String



Let's look at the real code

```
1 public Date(int theMonth, int theDay, int theYear) {
2     month = checkMonth(theMonth);
3     year = theYear;
4     day = checkDay(theDay);
5     System.out.printf(
6         "Date object constructor for date %s\n", this);
7 }
8
9 private int checkMonth(int testMonth) {
10     if(testMonth > 0 && testMonth <=12) return testMonth;
11     else {
12         System.out.printf(
13             "Invalid month (%d), set to 1", testMonth);
14         return 1;
15     }
16 }
```

<<Java Class>>


 **Date**

(default package)

▣ month: int


▣ day: int

▣ year: int

 **Date(int,int,int)**

▣ checkMonth(int):int

▣ checkDay(int):int

 **toString():String**



Let's look at the real code

```
17 private int checkDay(int testDay) { // data validation
18     int[] daysPerMonth = { 0, 31, 28, 31, 30, 31, 30, 31,
19         31, 30, 31, 30, 31 };
20     if(testDay > 0 && testDay <= daysPerMonth[month])
21         return testDay;
22     if(month == 2 && testDay == 29 && (year % 400 == 0 ||
23         (year % 4 == 0 && year % 100 != 0)))
24         return testDay;
25     System.out.printf("Invalid day (%d), set to 1",
26         testDay);
27     return 1;
28 }
29 public String toString() {
30     // transform object to String representation
31     return String.format("%d/%d/%d", month, day, year);
32 }
```

<<Java Class>>


 **Date**

(default package)

▣ month: int


▣ day: int

▣ year: int

 Date(int,int,int)

▣ checkMonth(int):int

▣ checkDay(int):int


 toString():String



Let's look at the real code

```
1 public class Employee {  
2     private String firstName;  
3     private String lastName;  
4     private Date birthDate;  
5     private Date hireDate;  
6 }
```

<<Java Class>>

 **Employee**


(default package)


▪ firstName: String

▪ lastName: String

▪ birthDate: Date

▪ hireDate: Date

 Employee(String,String,Date,Date)


 toString():String



Let's look at the real code

```
1 public Employee(String first, String last,
2   Date dateOfBirth, Date dateOfHire) { //
3   constructor
4   firstName = first;
5   lastName = last;
6   birthDate = dateOfBirth;
7   hireDate = dateOfHire;
8 }
9
10 public String toString() { // to String
11   representation
12   return String.format("%s, %s Hired: %s
13     Birthday: %s",
14     lastName, firstName, hireDate, birthDate
15   );
16 }
```

<<Java Class>>

 **Employee**


(default package)


▪ firstName: String

▪ lastName: String

▪ birthDate: Date

▪ hireDate: Date

 Employee(String,String,Date,Date)

 toString():String



Let's run the code

```
1 public class EmployeeTest {  
2     public static void main(String[] args) {  
3         Date birth = new Date(7, 24, 1949);  
4         Date hire = new Date(3, 12, 1988);  
5         Employee employee = new Employee("Bob", "Blue", birth, hire);  
6         System.out.println(employee);  
7     }  
8 }
```

Date object constructor for date 7/24/1949

Date object constructor for date 3/12/1988

Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949



A Time class

```
1 public class Time1 {
2     private int hour; // 0 - 23
3     private int minute; // 0 - 59
4     private int second; // 0 - 59
5     // set a new time value using universal time
6     public void setTime(int h, int m, int s) { ... }
7     // convert to String in universal-time format (HH:MM:SS)
8     public String toUniversalString() { ... }
9     // convert to String in standard-time format (H:MM:SS AM or PM)
10    public String toString() { ... }
11    public void setTime(int h, int m, int s) {
12        hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
13        minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
14        second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
15    }
16 }
```



A Time class

```
1 public class Time1 {  
2     // convert to String in universal-time format (HH:MM:SS)  
3     public String toUniversalString() {  
4         return String.format("%02d:%02d:%02d", hour, minute, second);  
5     }  
6  
7     // convert to String in standard-time format (H:MM:SS AM or PM)  
8     public String toString() {  
9         return String.format("%d:%02d:%02d %s",  
10             ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),  
11             minute, second, ( hour < 12 ? "AM" : "PM" ) );  
12     }  
13 }
```



toString() method

- `toString()` is one of the methods that every class inherits directly or indirectly from class `Object`.
 - Returns a `String` that “textually represents” an object.

```
1 Time1 time1 = new Time1();
```

- Called implicitly whenever an object must be converted to a `String` representation (e.g., `System.out.println(time1)`).
- `System.out.printf("%s\n", time1)`
is equivalent to
`System.out.printf("%s\n", time1.toString())`



Default constructor

- Class `Time1` does not declare a constructor.
- It will have a default constructor supplied by the compiler.
- `int` instance variables implicitly receive the default value `0`.
- Instance variables also can be initialized when they are declared in the class body, using the same initialization syntax as with a local variable.

```
1 public class Time1 {  
2     private int hour = 10; //default constructor will not initialize  
    hour  
3     private int minute; //default constructor will initialize minute to  
    0  
4     private int second; //default constructor will initialize second to  
    0  
5 }
```


Using the `Time` class

```
1 public class Time1Test {  
2     public static void main(String[] args) {  
3         Time1 time = new Time1(); // invoke default constructor  
4         System.out.print("The initial universal time is: ");  
5         System.out.println(time.toUniversalString());  
6         System.out.print("The initial standard time is: ");  
7         System.out.println(time.toString());  
8     }  
9 }
```

```
The initial universal time is: 00:00:00  
The initial standard time is: 12:00:00 AM
```



Manipulating the object

```
1 public class Time1Test {  
2     public static void main(String[] args) {  
3         Time1 time = new Time1();  
4         time.setTime(13, 27, 6);  
5         System.out.print("Universal time after setTime is: ");  
6         System.out.println(time.toUniversalString());  
7         System.out.print("Standard time after setTime is: ");  
8         System.out.println(time.toString());  
9     }  
10 }
```

```
Universal time after setTime is: 13:27:06  
Standard time after setTime is: 1:27:06 PM
```



Handling invalid values

- When receiving invalid values, we could also simply leave the object in its current state, without changing the instance variable.
 - `Time` objects begin in a valid state and `setTime` method rejects any invalid values.
 - Some designers feel this is better than setting instance variables to zeros.

```
1 // set a new time value using universal time
2 public void setTime(int h, int m, int s) {
3     if(h >= 0 && h < 24) hour = h; // reject invalid values
4     if(m >= 0 && m < 60) minute = m;
5     if(s >= 0 && s < 60) second = s;
6 }
```



Notifying the client code

- Approaches discussed so far do not inform the client code of invalid values (no return to callers).

```
1 // approach 1: setting to zeros
2 public void setTime(int h, int m, int s) {
3     hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
4     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
5     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
6 }
```

```
1 // approach 2: keeping the last object state
2 public void setTime(int h, int m, int s) {
3     if(h >= 0 && h < 24) hour = h;
4     if(m >= 0 && m < 60) minute = m;
5     if(s >= 0 && s < 60) second = s;
6 }
```



Notifying the client code

- `setTime` could return a value such as `true` if all the values are consistent and `false` if any of the values are invalid.
 - The caller would check the return value, and if it were `false`, would attempt to set the time again.
- In *exception handling*, we'll learn techniques that enable methods to indicate when invalid values are received.



this reference

- The keyword **this** is a reference variable that refers to the current object in Java.
- When a **non-static** method is called on a particular object, the method's body implicitly uses keyword **this** to refer to the object's instance variables and other methods.

```
1 public class Time1 {  
2     private int hour; // 0 - 23  
3     private int minute; // 0 - 59  
4     private int second; // 0 - 59  
5     // set a new time value using universal time  
6     public void setTime(int hour, int minute, int second) {  
7         if(hour >= 0 && hour < 24) this.hour = hour;  
8         if(minute >= 0 && minute < 60) this.minute = minute;  
9         if(second >= 0 && second < 60) this.second = second;  
10    }  
11 }
```



Overloaded constructors

- Method overloading: methods of the same name can be declared in the same class, as long as they have different sets of parameters.
 - Used to create methods that perform same tasks on different types or different numbers of arguments.
- Similarly, *overloaded constructors* enable objects of a class to be initialized in different ways (constructors are special methods).
- Compiler differentiates overloaded methods/constructors by their signature (method name, the type, number, and order of parameters).
 - `max(double, double)` and `max(int, int)`.

Overloaded constructors

```
1 public class Time2 {  
2     public Time2(int h, int m, int s) {  
3         setTime(h, m, s);  
4     }  
5     public Time2(int h, int m) {  
6         this(h, m, 0);  
7     }  
8     public Time2(int h) {  
9         this(h, 0, 0);  
10    }  
11    public Time2() {  
12        this(0, 0, 0);  
13    }  
14    public Time2(Time2 time) {  
15        this(time.getHour(), time.getMinute(), time.getSecond());  
16    }  
17 }
```



Using overloaded constructors

```
1 public class Time2Test {  
2     public static void main(String[] args) {  
3         Time2 t1 = new Time2();  
4         Time2 t2 = new Time2(2);  
5         Time2 t3 = new Time2(21, 34);  
6         Time2 t4 = new Time2(12, 25, 42);  
7         Time2 t5 = new Time2(27, 74, 99);  
8         Time2 t6 = new Time2(t4);  
9         System.out.println(t1.toUniversalString());  
10        System.out.println(t2.toUniversalString());  
11        System.out.println(t3.toUniversalString());  
12        System.out.println(t4.toUniversalString());  
13        System.out.println(t5.toUniversalString());  
14        System.out.println(t6.toUniversalString());  
15    }  
16 }
```

```
00:00:00  
02:00:00  
21:34:00  
12:25:42  
00:00:00  
12:25:42
```



More on constructors

- Every class must have **at least one constructor**.
- If you do not provide any constructors in a class's declaration, the compiler creates a **default constructor that takes no arguments** when it's invoked.
- The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values (zero for primitive numeric types, false for boolean values and null for references).
- If your class declares any constructors, the compiler will **not create a default constructor**.
 - In this case, you can't initialize objects with `new ClassName()` if your constructor requires input arguments.



Creating packages

- Each class in the Java API belongs to a package that contains a group of related classes.
- Packages help programmers manage the complexity of application components.
- Packages facilitate software reuse by enabling programs to import classes from other packages, rather than copying the classes into each program that uses them.
- Packages provide a convention for unique class names, which helps prevent class-name conflicts.

Creating packages

- 1 Declare a `public` class.
- 2 Choose a package name and add a `package` declaration to the source file for the reusable class declaration.
 - In each Java source file there can be **only one package declaration**, and it must precede all other declarations and statements.
 - `package course.cs102a;`



Creating packages

- Placing a package declaration at the beginning of a Java source file indicates that the class declared in the file is part of the specified package.
- A Java source file must have the following order:
 - A `package` declaration (if any);
 - `import` declarations (if any);
 - `class` declarations (you can declare multiple classes in one .java file, but only one `public`).
- **Only one of the class declarations in a particular file can be public.**
- Other classes in the file are placed in the `package` and can be used only by the other classes in the `package`. Non-public classes are in a package to support the reusable classes in the package.



Creating packages

- When a Java file containing a package declaration is compiled, the resulting class file is placed in the directory specified by the declaration.
- The class `Time1` should be placed in the directory `course/cs102a/`.
- `javac` command-line option `-d` causes the compiler to create appropriate directories based on the class' package declaration.
- Example command: `javac -d . Time1.java`
 - Specifies that the first directory in our package name should be placed in the current directory (`.`).
 - The compiled classes are placed into the directory that is named last in the package declaration.
 - `Time1.class` will appear in the directory `./course/cs102a/`.



Creating packages

- package name is part of the fully qualified name of a class, e.g., `course.cs102a.Time1`.
- We can use the fully qualified name in programs or `import` the class and use its simple name (e.g., `Time1`).
- If another package contains a class of the same name, the fully qualified class names can be used to distinguish between the classes in the program and prevent a name conflict.



Package access

- If no access modifier is specified for a class member when it's declared in a class, it is considered to have package access.

Modifier	Class	Package	World
public	Y	Y	Y
no modifier	Y	Y	N
private	Y	N	N



Type casting in Java

- Assigning a value of one type to a variable of another type requires type casting

```
1 int x = 10;  
2 byte y = (byte) x;
```

- In Java, type casting is classified into two categories:
 - Widening casting (implicit): `byte`→`short`→`int`→`long`→`float`→`double`.
 - Narrowing casting (explicitly done):
`double`→`float`→`long`→`int`→`short`→`byte`.



Type casting in Java

- Automatic Type casting take place when
 - the two types are compatible,
 - the target type is larger than the source type.

```
1 int x = 10;  
2 float y = x;
```

- When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

```
1 int x = 10;  
2 byte y = (byte) x;
```



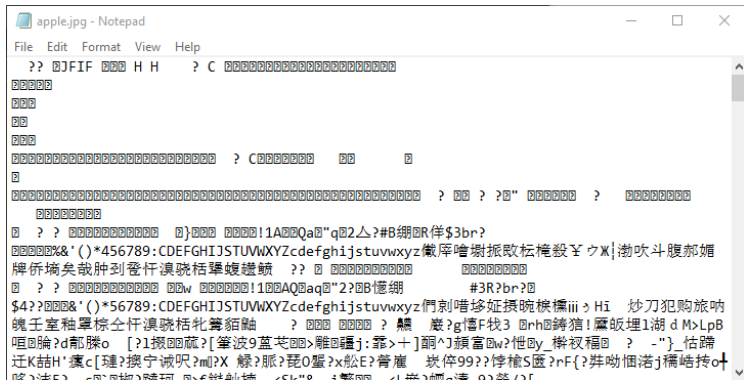
What is files in computer?

- Let say you have a file “apple.jpg” on your desktop, try open it by double clicking it. What do you see?



What is files in computer?

- Now try to open the file by notepad.exe, What do you see?



What is files in computer?

- Now try to open the file with Hex editor (e.g. notepad++.exe), What do you see?

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	ff	d8	ff	e0	00	10	4a	46	49	46	00	01	01	01	00	48	??JFIF.....H
00000010	00	48	00	00	ff	fe	00	3c	43	52	45	41	54	4f	52	3a	.H.. ?<CREATOR:
00000020	20	67	64	2d	6a	70	65	67	20	76	31	2e	30	20	28	75	gd-jpeg v1.0 (u
00000030	73	69	6e	67	20	49	4a	47	20	4a	50	45	47	20	76	38	sing IJG JPEG v8
00000040	30	29	2c	20	71	75	61	6c	69	74	79	20	3d	20	31	30	0), quality = 10
00000050	30	0a	ff	db	00	43	00	02	01	01	01	01	01	02	01	01	0. ?C.....
00000060	01	02	02	02	02	02	04	03	02	02	02	02	05	04	04	03
00000070	04	06	05	06	06	06	05	06	06	06	07	09	08	06	07	09
00000080	07	06	06	08	0b	08	09	0a	0a	0a	0a	0a	06	08	0b	0c
00000090	0b	0a	0c	09	0a	0a	0a	ff	db	00	43	01	02	02	02	02 ?C.....
000000a0	02	02	05	03	03	05	0a	07	06	07	0a	0a	0a	0a	0a	0a
000000b0	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a
000000c0	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a
000000d0	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	0a	ff	c0	00	11 ?
000000e0	08	01	39	01	38	03	01	22	00	02	11	01	03	11	01	ff	..9.8..".
000000f0	c4	00	1e	00	00	01	03	05	01	01	00	00	00	00	00	00	?.....
00000100	00	00	00	00	00	03	04	05	01	06	07	08	09	02	0a	ff
00000110	c4	00	4b	10	00	02	01	02	05	02	04	03	06	04	01	09	?K.....
00000120	06	04	07	00	01	02	03	04	11	00	05	06	12	21	07	31!.1



Text file

- Notepad.exe is supposed to open text file.
- It contains only the text entered by the user and does not include formatting.
- A text file in which each byte represents one character according to the ASCII code is also called a ASCII file.
- ASCII is not enough since we have a lot of more character to represent, e.g. Chinese character.
 - Thus we have other coding system.

Binary file

- There is no one-to-one mapping between bytes and characters.
- The file may be processed byte by byte, or x-byte by x-byte where x depends on the application.



Writing file

- Let say you want to save a value 7000000 to a file. Do you want to save in a binary file or text file ?
 - For a binary file, it needs 4 bytes (an integer value)
 - For a text file, it needs 7 bytes (7 ascii characters)
- In JAVA, you need to specify how you want to write into a file.



Sequential-Access Text Files

- In this section, you'll manipulate sequential-access text files in which records are stored in order by the record-key field.

Creating a Sequential-Access Text File

- Java imposes no structure on a file – concepts such as records do not exist as part of the Java language.
 - You must structure files to meet the requirements of your applications.
- In the following example, we see how to impose a keyed record structure on a file.
- The program in the next page creates a simple sequential-access file that might be used in an accounts receivable system to keep track of the amounts owed to a company by its credit clients.
 - Uses the account number as the record key—the file will be created and maintained in account-number order.
 - The program assumes that the user enters the records in account-number order.



Creating a Sequential-Access Text File

```
1 // Writing data to a sequential text file with class Formatter.
2 import java.util.Formatter;
3 import java.util.Scanner;
4
5 public class CreateTextFile {
6     public static void main(String[] args) throws Exception {
7         Formatter output = new Formatter("clients.txt"); // open the file
8         Scanner input = new Scanner(System.in); // reads user input
9
10        int accountNumber; // stores account number
11        String firstName; // stores first name
12        String lastName; // stores last name
13        double balance; // stores account balance
```



Creating a Sequential-Access Text File

```
14 System.out.printf("%s\n%s\n%s\n%s\n\n",
15     "To terminate input, type the end-of-file indicator ",
16     "when you are prompted to enter input.",
17     "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
18     "On Windows type <ctrl> z then press Enter");
19 System.out.printf("%s\n%s",
20     "Enter account number (> 0), first name, last name and balance.",
21     "? ");
22
23 while (input.hasNext()) {
24     //retrieve data to be output
25     accountNumber = input.nextInt(); // read account number
26     firstName = input.next(); // read first name
27     lastName = input.next(); // read last name
28     balance = input.nextDouble(); // read balance
```



Creating a Sequential-Access Text File

```
29     if (accountNumber > 0) {
30         // write new record
31         output.format("%d %s %s %.2f\n", accountNumber,
32             firstName, lastName, balance);
33     } else {
34         System.out.println("Account number must be greater than 0.");
35     }
36     System.out.printf("%s %s\n%s", "Enter account number (>0)",
37         "first name, last name and balance.", "? ");
38 }
39 output.close();
40 }
41 }
```



Creating a Sequential-Access Text File

To terminate input, type the end-of-file indicator when you are prompted to enter input.

On UNIX/Linux/Mac OS X type `<ctrl> d` then press Enter

On Windows type `<ctrl> z` then press Enter

Enter account number (>0), first name, last name and balance.

? 100 Bob Jones 24.98

Enter account number (>0), first name, last name and balance.

? 200 Steve Doe -345.67

Enter account number (>0), first name, last name and balance.

? 300 Pam White 0.00

Enter account number (>0), first name, last name and balance.

? 400 Sam Stone -42.16

Enter account number (>0), first name, last name and balance.

? ^Z



Creating a Sequential-Access Text File

clients.txt

Account	First Name	Last Name	Balance
100	Bob	Jones	24.98
200	Steve	Doe	-345.67
300	Pam	White	0.00
400	Sam	Stone	-42.16



Creating a Sequential-Access Text File

- A `Formatter` outputs formatted `Strings`, using the same capabilities as method `System.out.printf`.
- A `Formatter` can output to various locations, such as the screen or a file.
- The argument in parentheses to the right of `Formatter` is a `String` containing the name and location of the file in which the program will write text.
- If the location is not specified in the `String`, as is the case here, the JVM assumes that the file is in the directory from which the program was executed.
- For text files, we use the `.txt` file extension.
- If the file does not exist, it will be created.
- If an existing file is opened, its contents are *truncated* – all the data in the file is discarded.



Reading Data from a Sequential-Access Text File

- We store data in files so that it can be retrieved for processing when needed.
- This section shows how to read data sequentially from a text file using a [Scanner](#).
- We placed the applications from Figs. 6.30 and 6.33 in the same directory so that they can both access the same `clients.txt` file, which in each case is assumed to be in the same directory as the application.

Reading Data from a Sequential-Access Text File

```
1 // This program reads a text file and displays each record.
2 import java.io.File;
3 import java.util.Scanner;
4
5 public class ReadTextFile {
6     public static void main(String[] args) throws Exception {
7         // open the text file for reading with a Scanner
8         Scanner input = new Scanner(new File("clients.txt"));
9
10        int accountNumber; // stores account number
11        String firstName; // stores first name
12        String lastName; // stores last name
13        double balance; // stores account balance
14
15        System.out.printf("%-10s%-12s%-12s%10s\n", "Account",
16            "First Name", "Last Name", "Balance");
```



Reading Data from a Sequential-Access Text File

```
17 while (input.hasNext()) {  
18     accountNumber = input.nextInt(); // read account number  
19     firstName = input.next(); // read first name  
20     lastName = input.next(); // read last name  
21     balance = input.nextDouble(); // read balance  
22  
23     System.out.printf("%-10s%-12s%-12s%10.2f\n",  
24         accountNumber, firstName, lastName, balance);  
25 }  
26  
27 input.close();  
28 }  
29 }
```



Reading Data from a Sequential-Access Text File

Account	First Name	Last Name	Balance
100	Bob	Jones	24.98
200	Steve	Doe	-345.67
300	Pam	White	0.00
400	Sam	Stone	-42.16



Reading Data from a Sequential-Access Text File

- There are several ways to initialize a `Scanner`.
- When reading data from the user at the keyboard, we initialize a `Scanner` with the `System.in` object, which represents a stream that is connected to the keyboard.
- To read data from a file, you must initialize the `Scanner` with a `File` object.
- If the file cannot be found, an exception (of type `FileNotFoundException`) occurs and the program terminates immediately.
- We show how to handle such exceptions later.

Reading Directory Information

```
1 import java.io.File;
2
3 public class Listname {
4     public static void main(String a[]) {
5         File file = new File("C:\\Users\\todd\\Desktop");
6         String[] fileList = file.list();
7         for(String name : fileList) {
8             System.out.println(name);
9         }
10    }
11 }
```

