

Topic 6: Classes and Objects

Reading: JBD Chapter 6

Types in Java

- A type is defined by a set of values and a set of operations (operators or methods)
- Some types are part of the Java language
 - Primitive types: `int`, `double`, `boolean`, etc.
 - Reference types: `String`, `StringBuffer`, etc.
- `enum` types are simple user-defined types
 - Finite sets of values
- More complex user-defined types: Classes
 - An instance of a *class* is called an *object*
 - An object can have a complex internal state
 - A class can define *methods* that apply to individual objects

Example: a **Date** class

- An instance of the **Date** class is called a **Date** object.
- An object can have *instance variables* that define its state (also known as *instance fields*)

```
class Date {  
    int day;  
    int month;  
    int year;  
}
```

- You can use the class to declare a variable

```
Date d1;
```

- **Date** is a *reference type* (like **String**)
- The value of **d1** is a reference to a **Date** object

Constructing Objects

```
class Date {  
    int day;  
    int month;  
    int year;  
}
```

- To make a new **Date** object, use the **new** keyword and parentheses:

```
Date d1 = new Date();
```

- To access the instance variables inside a **Date** object, use dot notation:

```
d1.day  
d1.month  
d1.year
```

Initializing Instance Variables

```
class Date {  
    int day;  
    int month;  
    int year;  
}
```

- By default, the instance variables in a new object are initialized to values depending on their primitive type (just like the elements of an array)

```
Date d1 = new Date();  
System.out.println(d1.day + "/" + d1.month  
                   + "/" + d1.year);
```

0/0/0

Specifying Default Initial Values

- You can also provide your own initial values for some or all of the instance variables of a class:

```
class Date {
    int day;
    int month;
    int year = 2009;
}
Date d1 = new Date();
System.out.println(d1.day + "/" + d1.month
                  + "/" + d1.year);
```

0/0/2009

Assigning Values to Instance Variables

```
class Date {  
    int day;  
    int month;  
    int year;  
}
```

- You can assign values to instance variables by using assignment statements

```
Date d1 = new Date();  
d1.day = 25;  
d1.month = 12;  
d1.year = 2000;  
System.out.println(d1.day + "/" + d1.month  
                    + "/" + d1.year);
```

25/12/2000

Constructors

- A *constructor* is a special kind of method
 - Its name is the same as the name of its class
 - It does not return anything (not even `void`)
 - It is called whenever a new object is created (by `new`)
 - Its purpose is to initialize the new object
- Every class has at least one constructor
 - By default, Java provides a constructor that takes no parameters and does nothing
 - You can provide one or more constructors of your own (if you do, the Java-provided default constructor goes away)
 - Each constructor must have a unique signature
 - `Date(int y)`
 - `Date(int d, int m, int y)`

A Constructor for the Date class

```
class Date {
    int day;
    int month;
    int year = 2009;
    Date(int d, int m, int y) { //constructor
        day = d;
        month = m;
        year = y;
    } // end of constructor
} // end of class Date

Date d1 = new Date(15, 6, 2007);
System.out.println(d1.day + "/" + d1.month
                    + "/" + d1.year);
```

15/6/2007

Copy Constructors

- A *copy constructor* makes a new copy of an existing object

```
class Date {  
    int day;  
    int month;  
    int year;  
    Date(Date d) {  
        day = d.day;  
        month = d.month;  
        year = d.year;  
    }  
} // end of class Date
```

- Calling a copy constructor:

```
Date d1 = new Date(15, 6, 2007);  
Date d2 = new Date(d1);
```

Instance Methods

- If a method is not declared **static**, it is an *instance method*
- An instance method applies to one specific object (an instance of the class)

```
class Date {  
    int day;  
    int month;  
    int year;  
    String expectedTemperature() {  
        if (month >= 5 && month <= 9)  
            return "warm";  
        else return "cold";  
    } // end of expectedTemperature()  
} // end of class Date
```

Invoking an Instance Method

- To invoke an instance method, use these parts:
 - An expression that evaluates to an object (we will call this the *referenced object*)
 - Dot
 - A method call (with arguments, if needed)

```
Date d1 = new Date(15, 6, 2007);
String climate = d1.expectedTemperature();
System.out.println("The climate on "
    + d1.day + "/" + d1.month + "/" + d1.year
    + " is " + climate);
```

The climate on 15/6/2007 is warm

Printing Your Objects

- You can write an instance method named `toString()` for your class, that returns a String
- Construct the String any way you like
- Declare the `toString()` method **public** (this means that it can be called by methods of other classes)

```
class Date {
    int day;
    int month;
    int year;
    public String toString() {
        return day + "/" + month + "/" + year;
    } // end of toString()
} // end of class Date
```

Printing Your Objects (continued)

- If your class has a public `toString()` method, `print` and `println` will call it whenever they need to print an instance of your class

```
Date d1 = new Date(15, 6, 2007);  
String climate = d1.expectedTemperature();  
System.out.println("The climate on "  
    + d1 + " is " + climate);
```

```
The climate on 15/6/2007 is warm
```

Static Variables vs. Instance Variables

- A variable declaration can be inside a class but not inside any method
 - The scope of these variables is *all* methods of the class
- A class can have two kinds of variables:
- Static variables (declared **static**)
 - Apply to the class as a whole
 - Have only a single value for the whole class
 - Do not need to be "constructed"
- Instance variables (*not* declared **static**)
 - Apply to a specific *referenced object* (instance of the class)
 - Have a different value for every object
 - Are created by **new** and initialized by the class constructor

Static Variables

- Represent some information that is global to the class
- Example: **Math.PI**
- An example for the **Date** class: remember how many days are in each month, using an **int** array
(We ignore leap years in this simple example)

```
class Date {  
    static int[] monthDays = {0, 31, 28, 31,  
                               30, 31, 30, 31, 31, 30, 31, 30, 31};  
    int day;  
    int month;  
    int year;  
    // methods go here  
} // end of class Date
```

Another example of an instance method

- Write an instance method named `nextDay()`
- `nextDay()` returns a `Date` that is one day later than the object on which it is invoked
- Signature:

```
class Date {  
    Date nextDay() {  
        // body goes here  
    }  
} // end of class Date
```

Invoking the `nextDay()` method

- To invoke an instance method of the `Date` class, you need a `Date` object (the "referenced object")
- The `nextDay()` method constructs a new `Date` and returns it

- Invocation:

```
Date d1 = new Date(30, 5, 2008);  
Date d2 = d1.nextDay();  
Date d3 = d1.nextDay().nextDay();  
System.out.println("Today is " + d1);  
System.out.println("Tomorrow is " + d2);  
System.out.println("The day after that is " + d3);
```

- Expected output:

```
Today is 30/5/2008  
Tomorrow is 31/5/2008  
The day after that is 1/6/2008
```

Implementing the `nextDay()` method

```
class Date {
    // Declare some variables and methods here
    Date nextDay() {
        Date tomorrow =
            new Date(day + 1, month, year);
        if (tomorrow.day > monthDays[month]) {
            tomorrow.month++;
            tomorrow.day = 1;
        }
        if (tomorrow.month > 12) {
            tomorrow.year++;
            tomorrow.month = 1;
        }
        return tomorrow;
    } // end of nextDay()
} // end of class Date
```

Static Methods vs. Instance Methods

- A static method (declared **static**)
 - Applies to a class as a whole
 - Can take parameters and return results
 - Does not require an object to invoke it
 - Is usually invoked using a class name
`MyClass.myStaticMethod(myArgs)`
- An instance method (*not* declared **static**)
 - Applies to an individual object
 - Can take parameters and return results
 - Can be invoked only through an existing object:
`myObject.myInstanceMethod(myArgs)`
 - Takes the referenced object as an implicit parameter
(use **this** to refer to the referenced object if necessary)

Example of a static method for the **Date** class

- The `later()` method returns the later of two Dates

```
class Date {  
    // Declare some variables and methods here  
    static Date later(Date d1, Date d2) {  
        if (d1.year > d2.year) return d1;  
        else if (d1.year < d2.year) return d2;  
        else if (d1.month > d2.month) return d1;  
        else if (d1.month < d2.month) return d2;  
        else if (d1.day > d2.day) return d1;  
        else return d2;  
    } // end of later()  
} // end of class Date
```

Invoking a Static Method

- The `later()` method returns the later of two Dates

```
Date d1 = new Date(15, 9, 2005);  
Date d2 = new Date(8, 12, 2005);  
Date d3 = Date.later(d1, d2);  
System.out.println("The later date is " + d3);
```

- Expected output:

```
The later date is 8/12/2005
```

- Prefix the name of the static method with the name of the class (unless you are inside a method of this class)
- Note that `later()` does *not* construct a new `Date`
 - Instead, it returns one of its arguments (a reference to an existing `Date`)

Summary of methods and variables

- Static methods and variables
 - Must be explicitly declared **static**
 - Apply to a class as a whole, not to an object
 - Are usually invoked using the class name
 - Examples: `Date.monthDays`, `Date.later(d1, d2)`
- Instance methods and variables
 - Declared with no **static** keyword
 - Apply to an individual object
 - Can only be invoked through an existing object
 - Examples: `d1.year`, `d1.nextDay()`
- Constructors
 - Special methods that initialize the state of an object
 - Implicitly called when an object is created by **new**

Rules for referencing variables

- Referencing static variables
 - From methods of the same class:
no class name needed (but it's a good practice)
 - From methods of other classes: class name needed
- Referencing instance variables
 - Inside an instance method of the same class,
defaults to the instance variables of the referenced object.
Example: `year`
 - Anywhere else: an object must be explicitly specified.
Example: `d1.year`
 - To refer to the referenced object explicitly, use `this`.
Example: `this.year`

Example of the use of **this**:

- `sameMonth()` is an instance method
- It must be invoked on a specific **Date**:
`d1.sameMonth(d2)`
- It refers to **month** of the referenced **Date** and also to **month** of another **Date**
- It uses **this** for clarity

```
class Date {
    // various declarations go here
    boolean sameMonth(Date other) {
        if (this.month == other.month)
            return true;
        else return false;
    } // end of sameMonth()
} // end of class Date
```

Lifespan of an Object

- An object is created by the keyword `new`
- Its instance variables are initialized by a constructor
- The object remains in existence as long as some variable contains a reference to it
- Then its storage is reclaimed by the garbage collector

Every object gets these operators:

- **==** operator tests two object references for identity

```
Date d1 = new Date(30, 6, 2005);
Date d2 = new Date(30, 6, 2005);
Date d3 = d2;
d1 == d2          // false
d2 == d3          // true
```

- **instanceof** operator returns true if an object is an instance of a given class (and is not null)

```
Date d1 = new Date(30, 6, 2005);
d1 instanceof Date    // true
d1 = null;
d1 instanceof Date    // false
```

Every object gets this method:

- `public String toString()`

- Instance method, converts the object to a `String`
- Automatically invoked by `print`, `println`, and `concatenate`
- By default, the result is not very helpful

```
Date d1 = new Date(30, 6, 2005);  
System.out.println(d1);
```

`Date@7c218e`

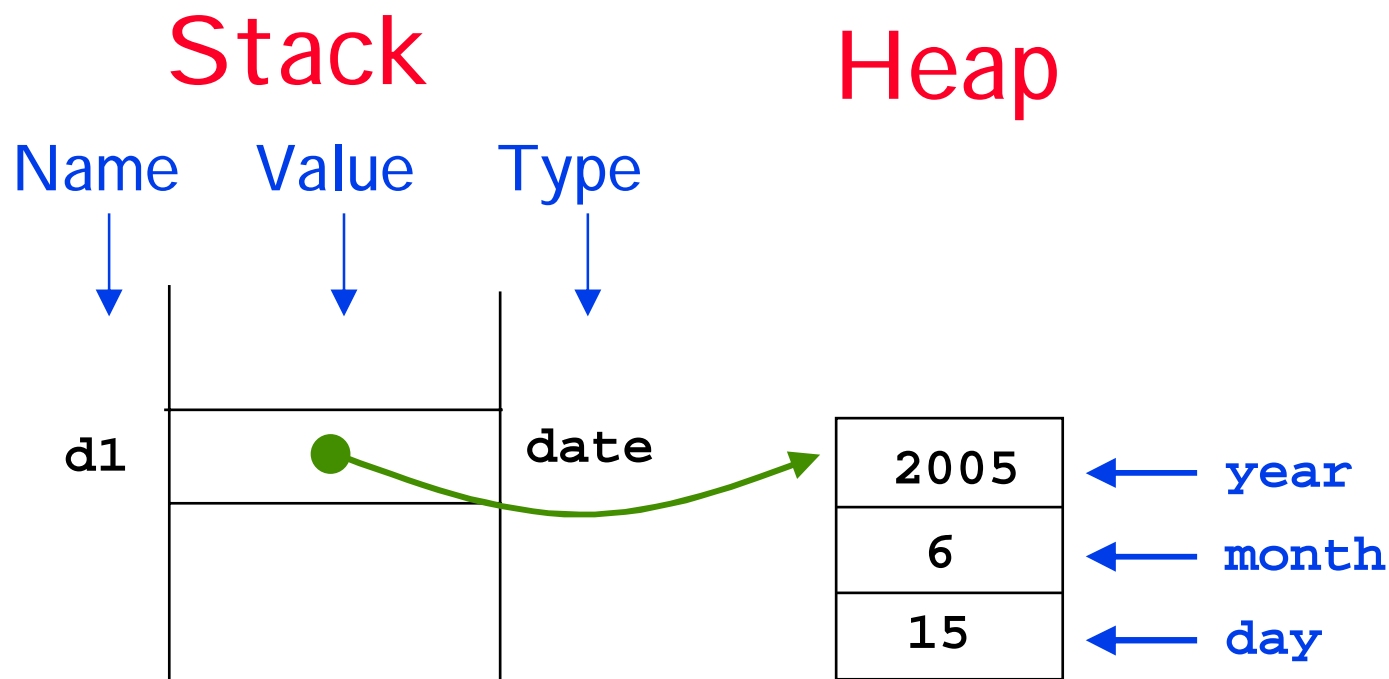
- But you can override it by writing your own `public toString()` method:

```
public String toString() {  
    return day + "/" + month + "/" + year;  
}  
System.out.println(d1);
```

`30/6/2005`

Passing Objects to Methods

- Classes are reference types
- When an object is passed to a method, the method gets a copy of the object reference (not the value)
- A method can change the value of an object (this is called *mutation*)



Packages, Classes, and Files

- A *package* is a collection of related classes
- A package can have a name
 - Example: `java.util`
 - Example: `edu.ucsc.cs12a.ourPackage`
- To access the classes in a package, use an import
 - Example: `import java.util.*`

Packages, Classes, and Files (continued)

- Some conventions:
 - For complex classes with many methods, put each class in a separate file with the same name as the class
 - For very simple classes like CS12A HW, it's OK to put multiple classes in one file (name the file after the class that has a `main` method)
 - It's legal to define a class inside another class, but we have no reason to do this in our simple examples
 - Put files that define related classes in the same directory (this makes them, by default, in the same package)

Packages, Classes, and Files (continued)

- To specify that your classes are in a named package, put a package statement at the top of your file:

```
package edu.ucsc.cs12a.ourPackage;
```

- By default: All the files in a directory define classes in the same (unnamed) package
- By default: A class is accessible to other classes in the same package.
- If you want your class to be used by code outside its package, declare it to be public:

```
public class Student {  
    // definition of class goes here  
}
```

Which is Correct?

- A Student enrolls in a School and takes Courses. In each Course, the Student receives a Grade. A Student has a Major, which has certain required Courses. If a Student receives a passing Grade in all the required Courses of his/her Major, the Student receives a Degree.
- A Student pays Tuition. Some Students also pay Fees for Room, Meals, and Parking. Some of these charges may be offset by Scholarships and Loans. Tuition and Fees must be paid before a Student is eligible to Register.
- A Student has a Height, Weight, Age, and Gender. A Student has a record of receiving certain Vaccinations on certain Dates. Some students have Conditions that require special Accommodations. Some students have a record of Treatments and Prescriptions.

Abstraction

- *Abstraction* is the process of defining the information about some object that is relevant for a particular purpose
 - Eliminating unnecessary details
 - Defining an interface to represent the relevant parts
 - Separating the interface from the implementation
- The "abstraction barrier" hides everything that the client does not need to know
- Hiding information behind an abstraction barrier is called *encapsulation*

Encapsulation in Java

- To encapsulate a class, expose only some of its methods
 - Hide instance variables
 - Hide unnecessary methods
- To hide a variable or method, declare it private

```
private int salary;
private void internalMethod(int parameter) {
    // body goes here
}
```
- Private variables and methods are accessible only to methods of the same class
- Applies to both static and instance variables and methods

Summary of Access Control Modifiers

- **private**: Accessible only within the same class
- **public**: Accessible to anybody
- default
(sometimes called "package" access, but no keyword):
Accessible only to classes in the same package
- (There's also a "protected" keyword, which we will discuss later.)

Where to use Access Control Modifiers

- Use them on the top-level members of a class (not on variables inside methods)

```
class Person {
    private static double SALARY_LIMIT = 100000;
    private double salary;
    public boolean giveRaise(double percent) {
        double newSalary;
        if (percent < 0 || percent > 50)
            return false;
        else {
            newSalary = salary * (1.0 + percent/100);
            if (newSalary > SALARY_LIMIT)
                return false;
            else salary = newSalary;
        }
        return true;
    } // end of giveRaise()
} // end of class Person
```

Here →

Not Here →

Advantages of Encapsulation

- Makes your classes easier to understand for users
- Allows you to change internals without affecting users
- Allows you to enforce consistency and integrity rules
 - Salary raises are between 0 and 50%
 - Salary is not greater than SALARY_LIMIT
 - Only registered students can receive a grade
 - If A is the spouse of B, then B is the spouse of A

Accessors and Mutators

- Methods that modify the state of an object are called *mutators*
- Methods that access the state of an object without modifying it are called *accessors*
- Accessors and mutators are useful in encapsulation

```
private String phoneNumber;  
public String getPhoneNumber();  
public void setPhoneNumber(String newNumber);
```

An example class: **Counter**

- Keeps a count of how many times it has been "clicked"
- Wraps back to zero when a limit is reached
- The limit is specified when the Counter is created

Implementing the Counter class

```
class Counter {
    int value;
    int limit;
    Counter(int limit) {    // constructor
        this.limit = limit; // note use of this
        value = 0;
    }
    void click() {
        value = (value + 1) % limit;
    }
} // end of class Counter
```

- Problem: What if a user does this?

```
Counter c = new Counter(100);
c.value = 500;
```

Encapsulating the Counter class

```
class Counter {  
    private int value;  
    private int limit;  
    public Counter(int limit) {    // constructor  
        this.limit = limit;    // note use of this  
        value = 0;  
    }  
    public int getValue() { ← Accessor  
        return value;  
    }  
    public void click() { ← Mutator  
        value = (value + 1) % limit;  
    }  
    public void reset() { ← Mutator  
        value = 0;  
    }  
} // end of class Counter
```

Using the Counter class

```
class CounterTest {
    public static void main(String[] args) {
        Counter c1 = new Counter(100);
        System.out.println(c1.getValue());           0
        for (int i = 1; i <= 99; i++)
            c1.click();
        System.out.println(c1.getValue());           99
        c1.click();
        System.out.println(c1.getValue());           0
        c1.click(); c1.click();
        System.out.println(c1.getValue());           2
        c1.reset();
        System.out.println(c1.getValue());           0
    } // end of main()
} // end of class CounterTest
```