

Revision

- In the previous few lectures, we discussed how to use arrays to store some data items, when the number of data items is known only at the time a program is executed.
- This allows us to write programs that are as complicated as we want. We have actually made a reverse-it program with it.
- Methods and local variables allow us to separate a big program into multiple independent parts, so that we can manage even moderately complicated programs.

0911B L18-1

Classes: a mechanism for further separations

- Methods are good because it allows you to group your program statements into independent units.
- To understand a program using a method, we only need to know the *intuitive meaning* of the method, instead of tracing all the individual statements. This saves us a lot of work.
- Analogously, we want to be able to separate the data similarly.
- Instead of thinking of some related data as the composition of the primitive type building them, we want to think it as one inseparable unit.

0911B L18-2

Example: big integer

- Suppose we want to represent an integer in Java that is too large for any of its primitive types.
- We can use an array of integers together with a character to represent it. The integers represent the digits, and the character represent the sign of the number. i.e.:

(char)sign - (int[])digits

2	0	4	4	9	8
---	---	---	---	---	---

- After that we can write methods to actually deal with such “big numbers”. Every time we spell out where we place the sign character and also where we place the digits.
- But then we need to know that a “big number” is formed by a sign and some digits in order to use it.

0911B L18-3

An abstraction of big numbers

- What we really want is to be able to create a big integer (say, from a String), manipulate with it (e.g. add two big integers) and convert it back to a String (so that it can be printed out).
- The less we know about how the big number is represented, the better. We want to isolate everything that need the knowledge of how the number is represented.
- Java support this by **Object**.
- An object contains some data fields (like sign, digits) and some methods to manipulate the fields (like add, subtract). To use it, we don't need to know how they are implemented.
- In this lecture we cover how to use an object. In the next week we learn how to implement an object.

0911B L18-4

Creating an Object

- From a user's point of view, an object is something created by calling the `new` operator with the name of a class. This creates an **instance** of the class.
- For example, Java has a built-in class called `java.math.BigInteger`. We can write it like `BigInteger` if we import `java.math.*`;
- To create a `BigInteger`, what we do is to evaluate an expression like `new BigInteger("200")`. We say we construct a `BigInteger`, using the `String` "200".
- Like arrays, this gives a reference that can be assigned to a reference. We assign the result to a variable of type `BigInteger`:

```
BigInteger b = new BigInteger("200");
```

0911B L18-5

Using an object

- Once we have a object reference referring to some created objects, we can make use of it by using its **instance methods**.
- In general, we “call” an instance method named `method` for an object by `obj.method(arglist)`, where `obj` is an object reference referring to that object. Just like other methods, we can give it a list of argument, and it can return a value.
- For example, if we have two `BigInteger`'s `a` and `b`, we can add them up to give a new `BigInteger` by `a.add(b)`.
- This creates a new `BigInteger` object, and return its reference.
- Similar methods exists for `subtract`, `multiply`, `divide`, `mod`, `max`, `min`, `gcd`, `pow`, etc.

0911B L18-6

More methods of BigInteger

- We can use `equals` to check whether two `BigInteger`'s `a` and `b` are equal, like `a.equals(b)`. This returns a boolean value `true` if they are equal, and `false` otherwise.
- We can use `compareTo` to compare two `BigInteger`'s `a` and `b`. For example, `a.compareTo(b)` returns `-1`, `0` and `1` depending on whether `a` is smaller, equal or larger than `b`.
- Finally, we can use `toString`, `doubleValue`, `intValue`, etc., to convert it to other types. E.g., if `num` is a reference to `BigInteger` storing a value equivalent to `1000`, then `num.toString()` returns `"2000"`, `num.intValue()` returns `2000`, and `num.doubleValue()` returns `2000.0`.

0911B L18-7

Example: finding factorial

- Let's say somehow we want to find the factorial of `400`. We cannot do it using even `long`: it cannot represent such huge values.
- We use `BigInteger`. Note that we don't need to know how it is implemented. In general, we don't need to know the implementation of a class to use it.

```
import chapman.io.*;
import java.math.*;

// Find the factorial of a number
public class Factorial {
    public static void main(String[] args) {
        StdIn in = new StdIn();
        System.out.print("Number? ");
        int num = in.readInt();
        BigInteger factorial = new BigInteger(""+1);
        for (int i = num; i > 1; --i)
            factorial = factorial.multiply(new BigInteger(""+i));
        System.out.println(num + "! = " + factorial.toString());
    }
}
```

0911B L18-8

Similarity with arrays

- Note that objects are very similar to arrays. Both are created with the `new` operator. Both uses a reference variable to refer to them. But the similarity does not end here...
- Objects are destroyed by the garbage collector, so they need not be explicitly destroyed.
- The reference of objects can be assigned among references of similar type. e.g. `BigInteger a = new BigInteger("250"); BigInteger b = a;`
- Comparisons of two references to objects check whether they refers to the same object, e.g. `a == b`.
- You can pass an object reference to a method, and an object reference can be returned from a method.

0911B L18-9

The mysterious StdIn

- Now we can understand better about what is `StdIn`.
- It is a class. When you write `new StdIn()`, it creates an object.
- Since it just creates an object without a name, you have to make a reference variable of type `StdIn` to hold the returned reference. So you write `StdIn in = new StdIn();`
- There are some fields of `StdIn` used for input. We don't need to know them to use a `StdIn` object.
- What we need to know is the instance methods

0911B L18-10

java.lang.String

- `String` is another type of Java objects. However, since `Strings` are so fundamental to Java programming, the language provide extra facility for dealing with `Strings`.
- We usually do not create a `String` using the `new` operator. Instead we only need to enclose something in double quotes to create a `String` (e.g. `"Hello World!"`).
- You can use the `+` operator to concatenate `String`'s, and if something of another type is added with it, that something is converted to `String` as well. E.g., `"Hello"+2` gives you `"Hello2"`.
- Such language support is only seen in the Java `String` class, not any other classes.

0911B L18-11

The class behaviour of String

- On the other hand, `String` is still a class. You can create it with `new`, you can access it through methods.
- We can create a `String` with a constructor. If we have a `char` array referred by `ch_arr`, we can make a new `String` by `new String(ch_arr)`. The created `String` will have exactly the same content as the `char` array.
- To check whether two `Strings` `a` and `b` are equal, you can use `a.equals(b)`. (Not `a == b`: it checks whether they point to the same object.)
- To compare two `Strings` `a` and `b`, you can use `a.compareTo(b)`. Again it returns a negative, `0` or positive values when `a` is smaller than, equal to and larger than `b` respectively.

0911B L18-12

Other interesting methods of String

- String provide a lot of useful methods (c.f. page 162 of text)
- Some more useful ones: suppose a and b are two String references,
 - `a.equalsIgnoreCase(b)` is similar to `a.equals(b)`, but uppercase and lowercase letters are considered to be the same.
 - `a.length()` gives you the length of a in number of char's.
 - `a.charAt(n)` gives you the (n+1)-st character of a.
 - `a.toLowerCase()` returns a new String that is the same as a except that all characters are in the lower case. Similar for `a.toUpperCase()`.

0911B L18-13

Trivial examples

Suppose `str1`, `str2` and `str3` are holding "Test", "tesT" and "Case".

- `str1.equalsIgnoreCase(str2)` gives
- `str1.equalsIgnoreCase(str3)` gives
- `str1.equals(str2)` gives
- `str1.toLowerCase()` gives
- `str1.length()` gives
- `str1.charAt(5)` gives

0911B L18-14

Example: counting the number of words in a String

For example, the following program counts the number of words in a String (boring parts omitted):

```
System.out.print("Type in a sentence: ");
String str = in.readString();
int num_words = 0;
boolean last_is_word_char = false;
for (int curr = 0; curr < str.length(); ++curr) {
    char curr_char = str.charAt(curr);
    if (!last_is_word_char && Character.isLetter(curr_char))
        ++num_words;
    last_is_word_char = Character.isLetter(curr_char);
}
System.out.println("It contains " + num_words + " words.");
```

0911B L18-15

More interesting example: java.util.GregorianCalendar

- The classes `String` and `BigInteger` are a little bit boring: they are "immutable" classes, meaning that you cannot never change the values of the objects. You can only change the reference.
- Let's look at a more interesting class, `java.util.GregorianCalendar`. (Abbreviated as `GregorianCalendar` if you import `java.util.*`).
- It represent a calendar date (and in fact also time).
- You create a `GregorianCalendar` object by specifying the year, month and day-of-month numbers in the constructor, like `new GregorianCalendar(2000, 10, 10)` (N.B.: January is 0. You can write 10 like `Calendar.NOVEMBER` to avoid confusion.)

0911B L18-16

Accessing a GregorianCalendar

- In order to use the `GregorianCalendar`, we need to know its instance methods. Here are some examples:
- Given two references to `GregorianCalendar` `date1` and `date2`,
 - `date1.before(date2)`: returns true if `date1` is before `date2`, and false otherwise. Similar for `date1.after(date2)`.
 - `date1.equal(date2)`: returns true if the two dates represent the same date.
 - `date1.get(FIELD)`: get the field of the date. Field is a constant defined in the class `Calendar`, like `Calendar.YEAR`, `Calendar.DAY_OF_WEEK`, `Calendar.DAY_OF_YEAR`, etc.
 - `date1.add(FIELD, value)`: add `value` to the field of the date, and automatically normalize it.

0911B L18-17

Trivial examples

Suppose we have

```
GregorianCalendar today = new GregorianCalendar(2000, 10, 10);
GregorianCalendar examdate = new GregorianCalendar(2001, 0, 3);
```

- `today.before(examdate)` returns
- `today.equals(examdate)` returns
- `today.get(Calendar.DAY_OF_WEEK)` returns
- `today.add(Calendar.DATE, 1)` returns

0911B L18-18

A possible way to print out the calendar

Skipping boring stuffs as usual.

```
System.out.print("Year? ");
int year = in.readInt();
System.out.print("Month? ");
int month = in.readInt()-1;

GregorianCalendar day1 = new GregorianCalendar(year, month, 1);
GregorianCalendar curr = new GregorianCalendar(year, month, 1);

while (curr.get(Calendar.DAY_OF_WEEK) != Calendar.SUNDAY)
    curr.add(Calendar.DATE, -1);

while (curr.before(day1) || curr.get(Calendar.MONTH) == month) {
    for (int i = 0; i < 7; ++i) {
        if (curr.before(day1))
            System.out.print(" ");
        else if (curr.get(Calendar.MONTH) == month)
            Fmt.printf("%2d ", curr.get(Calendar.DAY_OF_MONTH));
        else
            break;
        curr.add(Calendar.DATE, 1);
    }
    System.out.println();
}
```

0911B L18-19

Conclusion

- We usually want to group data that are logically related into one variable. E.g. big numbers, string, date.
- To do this, we want to have a class correspond to it. E.g., BigInteger, String, GregorianCalendar.
- The user of such classes do not need to know the actual representations of such objects (e.g. sign, length, month).
- We still don't know where such representation is specified. They are specified when we define a class—next lecture.

0911B L18-20