

Revision: instance methods

- A class without instance method can be useful because a reference to an object of that class can be passed from or to a method.
- This allows multiple primitive values to be returned from a method, and allow the method to modify the values within the object.
- However, most classes becomes useful when they are provided with instance methods.
- An instance method is a method that will always be called for an object of a particular class.

0911B L21-1

What is meant by “calling a method for an object”

An instance method is always called for an object like `card.toString()`. For example:

```
class PlayingCard {
    ...
    public String toString() {
        ...
    }
}

public class TestCard {
    public static void main(String[] args) {
        PlayingCard card = new PlayingCard();
        card.suit = 3;
        card.value = 9;
        System.out.println(card.toString());
    }
}
```

Note that `toString` is called like `card.toString()`: it calls `toString` for `card`. The problem is, the `toString` method need to have a way to refer to `card` (it can't say `card`: it don't know who call it)!

0911B L21-2

The this reference

- The `this` reference is the way for an instance method to know the object. It is very much like that the instance method always has a hidden argument called `this`.
- It is done completely automatically. You don't need to do anything, the compiler automatically make that “formal-argument” in the method if you omit `static`.
- You can thus write `this.suit` to get the suit, or `this.toString()` to call `toString()` for it.
- But you don't even need that: within an instance method, `suit` means `this.suit`, `toString()` means `this.toString()` etc. And again `this` is completely automatic.

0911B L21-3

Data invariant

- An object usually represent some real-world or abstract entity.
- It is usually desirable that the object will represent such an entity under any situation.
- For example, if we make a playing card object, it is usually undesirable that someone dealing with a playing card object can change it to Spade 15.
- We call such guarantees “data invariant”, meaning that “after a class method complete executing, the data is always like ...”
- For example, let's make our target on the `PlayingCard` object to give the following data invariant: the suit is always 0–3, and the value is always 2–14.

0911B L21-4

Establishing an invariant: constructors

- If a class want to make such a guarantee, it must not depend on the user of the class to initialize its fields. **The fields of an object must be initialized when it is created.**
- A constructor do exactly that. Constructors are always called when an object is created.
- Constructors looks exactly like other instance methods, except:
 1. It has no return type (no one is going to get that, anyway).
 2. It has the name of the class.
- If you do not define a constructor, the compiler automatically generate one that take no argument and do nothing.

0911B L21-5

Constructors for PlayingCard

Let's say we want to have a constructor that, given two numbers, it initialize the data fields. It adjust the values if they are out of range.

```
// Create a card, clamping at 0<=suit<=3 and 2<=value<=14
public PlayingCard(int a_suit, int a_value) {
    if (a_suit < 0)
        a_suit = 0;
    if (a_suit > 3)
        a_suit = 3;
    if (a_value < 2)
        a_value = 2;
    if (a_value > 14)
        a_value = 14;
    suit = (byte) a_suit;
    value = (byte) a_value;
}
```

Two things happens: (1) you can now call `new PlayingCard(3, 9)` in order to create a club 9; (2) you can no longer use `new PlayingCard()` to create a “empty” card—default constructor is no longer generated.

0911B L21-6

Maintaining the invariant

Even if we create the card this way, we still cannot guarantee that the field will never be invalid: the user can still access the fields directly and say something like `card.value = 15;`

Of course, that is just the mistake of the user, but a very difficult one to trace: the user don't understand the implementation!

To make sure that cannot happen, we stop the main program from doing anything with the fields directly.

Instead, we provide enough ways for the main program to use the object. For example, the program might want to know what is the value and suit of the card. So we can provide instance methods for that.

Doing this has an interesting advantage: if later you want to change the way the object is represented, *the main program need no change— as long as you provide exactly the same instance methods!*

0911B L21-7

Step 1: restricting access

The first step in maintaining the invariant is to restrict access.

- The `public` keyword is called an **access control modifier**: it specifies that everybody can access the field.
- Now we want the reverse: only a method within the class can access the field. So instead of writing `public`, we write `private`, meaning exactly that:

```
class PlayingCard {
    ...
    private byte suit;
    private byte value;
}
```

Traditionally, all such fields are put to the end of the class, since this is the part of the class that nobody need to look at.

0911B L21-8

Access specifiers

There are 4 types of access in Java. From the most restrictive:

- **Class access**: the variable, field or method can be used by any method within the same class. Specified by `private`.
- **Package access**: the variable, field or method can be used by any method within the same package. This is the default.
- **Protected access**: the variable, field or method can be used by any method of the same class and all "subclasses" (to be taught later). Specified by `protected`.
- **Public access**: the variable, field or method can be used by any method. Specified by `public`.

0911B L21-9

Adding more instance methods

Now there is no way for the main program to change the fields in a card. So it is an immutable class—the easiest way to maintain a data invariant.

But the program still want to know what is the suit/value of a card. We make instance methods for that.

```
// Return the suit of the card
public byte getSuit() {
    return suit;
}
// Return the value of the card
public byte getValue() {
    return value;
}
```

Now, if the main program has a `PlayingCard` object referred by `card`, it can get its properties, but not change it to something bad (or change it at all). Bingo!

0911B L21-10

Example: Black Jack

Now we have most of the things in our hands. Let's make an application using class: `play Black Jack`.

Black Jack is a really simple game:

- The dealer and the player try to get cards from a shuffled deck to compete for the highest "total point".
- However, don't get too much: one getting over 21 points lose.
- A card scores its face value if it is below 10. If it is above 10, then it scores 10.
- Except that an Ace scores either 1 or 11. Basically, if 11 won't cause the total score to be over 21, it scores 11. Otherwise it scores 1.

0911B L21-11

Conduct of Black Jack

The game follows the sequence below:

- The dealer draws a card and show it.
- Then two cards are given to the player.
- The player then requests as many cards as he like, one by one.
- If the player gets too much, he lose.
- Otherwise, the dealer get cards, until he score 16 or more.
- If dealer gets too much, he lose. Otherwise, the one getting more points wins.

0911B L21-12

Breaking down to classes

There are two things that we might want to use again in another program, so it is reasonable to expect 3 classes:

- A main program class, responsible for the game conduct and Black Jack score counting. (Not reusable: other games use completely different rules.)
- A `PlayingCard` class. Highly reusable, since all card games use it. We have already done that.
- An object that represent a shuffled deck of cards, that can give out a card every time it is requested to. Quite reusable: most card games uses a well-shuffled deck of cards.

In many cases, the decision to make one more class is because you envision that it can be used in other programs.

0911B L21-13

The class `RandomDeck`

- We want a class that represent a random deck. At the beginning it represents a deck full of cards, and then the main program can ask it to give a card.
- It should never give the same card twice.
- Let's design the `RandomDeck` class so that:
 - It takes no argument to construct. Once constructed, it holds 52 cards, randomized.
 - The program can call `getCard` to get a new card. It always returns a card that has never been returned by the same `RandomDeck` object. If no card remains, it returns `null`.
 - The program can call `numRemain` to know how many cards are still there.

0911B L21-14

Data fields

Let's represent a deck by an array of `PlayingCard`.

We also need to know what cards have been given to the main program already.

Let's just use a counter. For convenience, we distribute cards from the last card, so that the counter also serves for keeping the number of cards still in the deck.

So the class looks like the following:

```
class RandomDeck {
    ...
    private PlayingCard[] cards;
    private int num_remaining_cards;
}
```

0911B L21-15

Constructor

Now we need to construct the object: filling cards, `num_remaining_cards` and also shuffling it. We have done similar task before, so it is easy this time.

```
class RandomDeck {
    // Create a well shuffled deck
    public RandomDeck() {
        num_remaining_cards = 0;
        cards = new PlayingCard[52];
        for (int i = 0; i <= 3; ++i)
            for (int j = 2; j <= 14; ++j)
                cards[num_remaining_cards++] = new PlayingCard(i, j);
        shuffleCards();
    }
    // Internal method to shuffle a deck
    private void shuffleCards() {
        for (int i = cards.length-1; i >= 1; --i) {
            int rand = (int)(Math.random() * (i+1));
            PlayingCard rand_card = cards[rand];
            cards[rand] = cards[i];
            cards[i] = rand_card;
        }
    }
    ...
}
```

0911B L21-16

Other methods

There are two other methods for the class: one to give a card (`getCard`), one return the number of remaining cards (`numRemain`). Both are very simple methods.

```
// Give out a card
public PlayingCard getCard() {
    ...
}

// Check how many cards remains
public int numRemain() {
    ...
}
```

Once again, we are using a method (`numRemain`), instead of allowing direct access to the `num_remaining_card` variable.

0911B L21-17

Static's in a class

- Even in a class other than the main program class, we can have static class variables and static class methods.
- Recall that they are those variables with duration throughout the whole execution, and those methods that are called without an object.
- Why writing the method in a class other than the main class?
- As we will see, a class is usually put in a separate file. Keeping everything related to a class within that class will make it easy to handle the class (i.e., no need to go through the main program to do cut and paste).

0911B L21-18

The main method

- Not just the main program can have a `main` method. Every class can have its own `main`.
- This is usually used for testing the class.
- Compiling a file with multiple classes will give you multiple `.class` files, i.e., multiple classes.
- To run the `main` method for a class, e.g. `RandomDeck`, we type `java RandomDeck`.
- If `RandomDeck` does not have a method with name `main` taking a `String[]` argument, the Java interpreter will complain.

0911B L21-19

Testing RandomDeck

We can do a simple test: to create a deck and print out everything there.

```
class RandomDeck {  
    ...  
    // Test program for this class  
    public static void main(String[] args) {  
  
    }  
    ...  
}
```

One note here: we say `System.out.println(card)`. The Java compiler (actually, the routine printing objects) will call `toString` for us.

0911B L21-20

Alternative programming model

- Now it becomes apparent that we program in a quite different way if we use objects.
- The primary difference is that we first do a decomposition job: find out what can be reused by other programs.
- Then we find out a class that does not need other class, and completely implement that. Of course, we test the program in the way.
- Implementation of each class is done through top-down design as usual, and if a method becomes too complicated, we can always add a private method.
- This process is repeated as often as needed, until the whole program is completed.

0911B L21-21

The myth about public class

- In the last lecture, you learn that we have to define a class as non-public because one file can only contain one public class.
- For some classes that is used only used by a single class (or package), this works perfectly.
- But for classes that we want to reuse in other projects, this is not good.
- What we can do is to separate different classes into different files. So for the `BlackJack` program, we will use three different files to hold it: `PlayingCard.java`, `RandomDeck.java`, `BlackJack.java`.
- If someone else want to use our `PlayingCard` class, all we need is to give them the `PlayingCard.class` file and perhaps the `PlayingCard.java` file.

0911B L21-22