# Theory: toString()



19 minutes 0 / 5 problems solved



## §1. Using default toString

The root Java class `Object` has the `toString()` method to get the string representation of an object. If you'd like to have a string representation, override this method in your class.

First, let's consider an example based on the default `toString()` implementation provided by the `Object` class.

This is the `Account` class. It has three fields and one constructor.

```
class Account {

    private long id;
    private String code;
    private Long balance;

    public Account(long id, String code, Long balance) {
        this.id = id;
        this.code = code;

        this.balance = balance;

    }


    // getters and setters

}
```

Let's create an instance of the class and get the string representation of that instance:

```
Account account = new Account(1121, "111-123", 400_000L);


String accString = account.toString(); // org.demo.example.Account@27082746
```

A string like `org.demo.example.Account@27082746` is not exactly what we would like to see. What we got here is the full class name and the hashcode of the object. This is the default behavior of the method `toString()`.

## §2. Overriding toString when declaring a class

If we want to include fields in the string representation of an object, we should override the standard behavior of the `toString()` method.

Here is another version of the `Account` class where we've overridden the `toString()` method:

Current topic:

toString()

Topic depends on:

- ✕ Hiding and overriding
- ✕ The Object class

Table of contents:

```
class Account {

    private long id;
    private String code;
    private Long balance;

    public Account(long id, String code, Long balance) {
        this.id = id;
        this.code = code;

        this.balance = balance;

    }



    // getters and setters



    @Override

    public String toString() {

return "Account{id=" + id + ",code=" + code + ",balance=" + balance + "}";

    }

}
```

Let's create an instance of this class and get the string representation of the instance:

```
Account account = new Account(1121, "111-123", 400_000L);

String accString = account.toString(); // Account{id=1121,code=111-123,balance=400000}
```

Compared to the default string representation, this one gives us more information about the object and its attributes.

String representations are very useful for debugging and logging. You can use the `toString()` method to display a string representation of an object in the standard output:

```
// option 1
System.out.println(account.toString());

// option 2
System.out.println(account);
```

Some modern IDEs, such as *IntelliJ IDEA*, allow generating the overridden `toString()` method automatically. This is very convenient if your class has a lot of fields.

## §3. Overriding toString when subclassing

If you have a class hierarchy you can also override `toString()`.

Here is a hierarchy of two classes:

- `Person` with a single string field `name`;
- `Employee` that extends `Person` and adds the field `salary`.

```
class Person {

    protected String name;

    public Person(String name) {
        this.name = name;
    }

    @Override

    public String toString() {

        return "Person{name=" + name + "}";

    }

}



class Employee extends Person {



    protected long salary;



    public Employee(String name, long salary) {

        super(name);

        this.salary = salary;

    }



    @Override

    public String toString() {

        return "Employee{name=" + name + ",salary=" + salary + "}";

    }

}
```

It is considered to be a good practice to include the class name in the string representation when working with hierarchies.

Let's create objects of these two classes and print them as strings:

```
Person person = new Person("Helena");
Employee employee = new Employee("Michael", 10_000);

System.out.println(person);   // Person{name=Helena}
System.out.println(employee); // Employee{name=Michael,salary=10000}
```

## §4. Possible problems when overriding toString

Overriding the `toString()` method so far looks very simple but what if your class has another class as a type of a field? Sometimes it may cause an error.

See the following example with `Person` and `Passport` classes. We do not include getters and setters in the code to make it more compact.

```
class Person {

    private String name;
    private Passport passport;

    // getters and setters

    @Override
    public String toString() {

        return "Person{name='" + name + ",passport=" + passport + "}";

    }

}



class Passport {



    private String country;

    private String number;



    // getters and setters



    @Override

    public String toString() {

        return "Passport{country=" + country + ",number=" + number + "}";

    }

}
```

If a person has no passport (`null`), the string representation will contain `null`.

Here is an example of two objects.

```
Passport passport = new Passport();
passport.setNumber("4343999");
passport.setCountry("Austria");

Person person = new Person();
person.setName("Michael");
System.out.println(person);  // first print

person.setPassport(passport);

System.out.println(person);  // second print
```

This code prints:

```
  Person{name=Michael,passport=null} // first print

Person{name=Michael, passport=Passport{country=Austria, number=4343999}} // second
 print
```

It works very well, no problems here! But what if the passport has the backward reference to the person and tries to get the string representation of the person?

Let's add the following field and the corresponding setter to the class `Passport` :

```
private Person owner;
```

Let's also modify the `toString()` method as follows:

```
@Override
public String toString() {

 return "Passport{country=" + country + ",number=" + number + ",owner=" + owner
+ "}";
}
```

When we create two objects let's set the owner to the passport:

```
passport.setOwner(person);
```

Now we get the big problem — the program tries to get the string representation of the person that includes the string representation of passport that includes the string representation of the person. It causes `java.lang.StackOverflowError`.

There are several ways to fix this situation:

- do not include fields represented by your classes in the `toString()` method;
- exclude the field in the `toString()` method from one of the classes.

So, be careful when including fields in the `toString` method. Consider references between classes. If you don't need any information, it's better to exclude it. It will save you from fatal mistakes in the long run.

Report a typo

**120** users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (3) Hints (0) Useful links (0) Show discussion