Utilities (Part 2)

Implementing static features

Goals for Today

- learn about preventing class instantiation
- learn about methods
  - static methods
  - method header
  - method signature
  - method return type
  - method parameters
  - pass-by-value

`new Yahtzee` Objects

- our **`Yahtzee`** API does not expose a constructor
  - but

```
Yahtzee y = new Yahtzee();
```

    is legal

- if you do not define any constructors, Java will generate a default no-argument constructor for you
  - e.g., we get the **`public`** constructor

```
public Yahtzee() { }
```

    even though we did not implement it

Preventing Instantiation

- our **`Yahtzee`** API exposes only **`static`** constants (and methods later on)
  - its state is constant
- there is no benefit in instantiating a **`Yahtzee`** object
  - a client can access the constants (and methods) without creating a **`Yahtzee`** object

```
boolean hasTriple = Yahtzee.isThreeOfAKind(dice);
```

- can prevent instantiation by declaring a **`private`** constructor

```
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class Yahtzee {

  private Yahtzee() {
    // private and empty by design
  }

  public static final int NUMBER_OF_DICE = 5;

}
```

```
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class Yahtzee {

  private Yahtzee() {
    // throwing an exception prevents accidental instantiation
    throw new AssertionError();
  }

  public static final int NUMBER_OF_DICE = 5;

}
```

`private`

- **`private`** fields, constructors, and methods cannot be accessed by clients
  - they are not part of the class API

- **`private`** fields, constructors, and methods are accessible only inside the scope of the class

- a class with only **`private`** constructors indicates to clients that they cannot use **`new`** to create instances of the class

Methods

- a method performs some sort of computation
- in the previous lecture, we studied how to determine if a list of 5 dice represented a roll of 3-of-a-kind:

```
// dice is a List<Die> reference
Collections.sort(dice);
boolean isThreeOfAKind =
  dice.get(0).getValue() == dice.get(2).getValue() ||
  dice.get(1).getValue() == dice.get(3).getValue() ||
  dice.get(2).getValue() == dice.get(4).getValue();
```

- we can encapsulate this computation as a method in our utility class

```
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class Yahtzee {

  private Yahtzee() {
    // private and empty by design
  }

  public static final int NUMBER_OF_DICE = 5;

  public static boolean isThreeOfAKind(List<Die> dice) {
    List<Die> copy = new ArrayList<Die>(dice);
    Collections.sort(copy);
    boolean result = copy.get(0).getValue() == copy.get(2).getValue() ||
                     copy.get(1).getValue() == copy.get(3).getValue() ||
                     copy.get(2).getValue() == copy.get(4).getValue();
    return result;
  }

}
```

Why make a copy? Because we shouldn't change the client's dice.

Method header

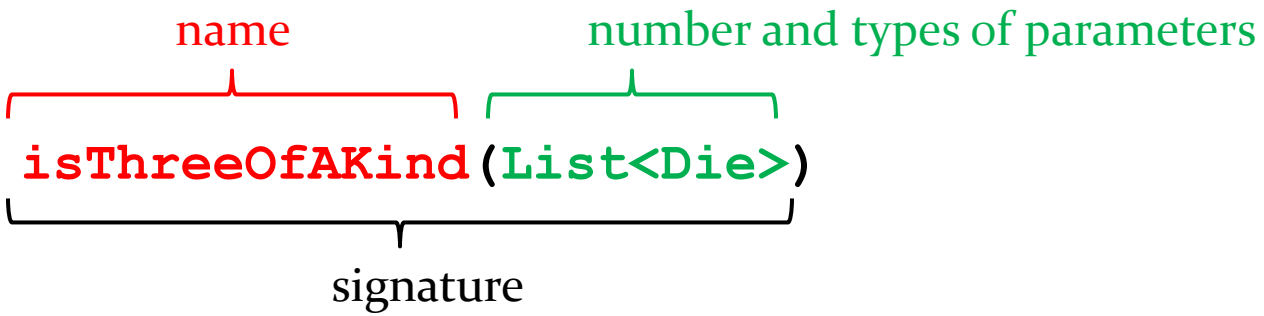
- the first line of a method declaration is sometimes called the *method header*

Method signature

- every method has a *signature*
  - the signature consists of the method name and the types in the parameter list
- our method

```
public static boolean isThreeOfAKind(List<Die> dice)
```

has the following signature



Method signature

- other examples from java.lang.String
  - headers
    - `String toUpperCase()`
    - `char charAt(int index)`
    - `int indexOf(String str, int fromIndex)`
    - `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
  - signatures
    - `toUpperCase()`
    - `charAt(int)`
    - `indexOf(String, int)`
    - `getChars(int, int, char[], int)`

Method Signatures

- method signatures in a class must be unique
- what happens if we try to introduce a second method

```
public static boolean
  isThreeOfAKind(Collection<Integer> dice)
```

?

- what about

```
public static boolean
  isThreeOfAKind(List<Integer> dice)
```

?

Method return types

- all Java methods return nothing (**`void`**) or a single type of value
- our method

```
public static boolean isThreeOfAKind(List<Die> dice)
```

has the return type **`boolean`**

Method return values

- if the method header says that a type is returned, then the method must return a value having the advertised type back to the client
- you use the keyword **`return`** to return the value back to the client

```
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class Yahtzee {

  private Yahtzee() {
    // private and empty by design
  }

  public static final int NUMBER_OF_DICE = 5;

  public static boolean isThreeOfAKind(List<Die> dice) {
    List<Die> copy = new ArrayList<Die>(dice);
    Collections.sort(copy);
    boolean result = copy.get(0).getValue() == copy.get(2).getValue() ||
                     copy.get(1).getValue() == copy.get(3).getValue() ||
                     copy.get(2).getValue() == copy.get(4).getValue();
    return result;
  }

}
```

Return the value of **`result`** back to the client.

Method return values

- a method stops running immediately after a return statement is run
  - this means that you are not allowed to have additional code after a return statement

Method parameters

- sometimes called *formal parameters*
- parameters are the variables that appear in the parameter list
- our method

```
public static boolean isThreeOfAKind(List<Die> dice)
```

has the parameter **`dice`**

Method parameters

- for a method, the parameter names must be unique
  - but a parameter can have the same name as a field (see [notes 1.3.3])
- the scope of a parameter is the body of the method
  - you can use the parameter just like you would any other variable inside the body of the method
  - the parameter does not exist outside of the method body

```
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class Yahtzee {

  private Yahtzee() {
    // private and empty by design
  }

  public static final int NUMBER_OF_DICE = 5;

  public static boolean isThreeOfAKind(List<Die> dice) {
    List<Die> copy = new ArrayList<Die>(dice);
    Collections.sort(copy);
    boolean result = copy.get(0).getValue() == copy.get(2).getValue() ||
                     copy.get(1).getValue() == copy.get(3).getValue() ||
                     copy.get(2).getValue() == copy.get(4).getValue();
    return result;
  }

}
```

Use the parameter **dice** to make a copy.

`static` Methods

- a method that is **`static`** is a per-class member
  - client does not need an object to invoke the method
  - client uses the class name to access the method

```
boolean hasTriple = Yahtzee.isThreeOfAKind(dice);
```

  - **`static`** methods are also called *class methods*
  - a **`static`** method can only use **`static`** fields of the class

[notes 1.2.4], [AJ 249-255]

Invoking Methods

- a client invokes a method by passing arguments to the method
  - the types of the arguments must be compatible with the types of parameters in the method signature
  - the values of the arguments must satisfy the preconditions of the method contract [JBA 2.3.3]

```
List<Die> dice = new ArrayList<Die>();
for (int i = 0; i < 5; i++) {
  dice.add(new Die());
}
boolean hasTriple = Yahtzee.isThreeOfAKind(dice);
```

argument

Pass-by-value

- Java uses pass-by-value to:
  - transfer the value of the arguments to the method
  - transfer the return value back to the client

- consider the following utility class and its client…

```
import type.lib.Fraction;

public class Doubler {

  private Doubler() {
  }

  // tries to double x
  public static void twice(int x) {
    x = 2 * x;
  }

  // tries to double f
  public static void twice(Fraction f) {
    long numerator = f.getNumerator();
    f.setNumerator( 2 * numerator );
  }
}
```

```
import type.lib.Fraction;

public class TestDoubler {

  public static void main(String[] args) {
    int a = 1;
    Doubler.twice(a);

    Fraction b = new Fraction(1, 2);
    Doubler.twice(b);

    System.out.println(a);
    System.out.println(b);
  }

}
```

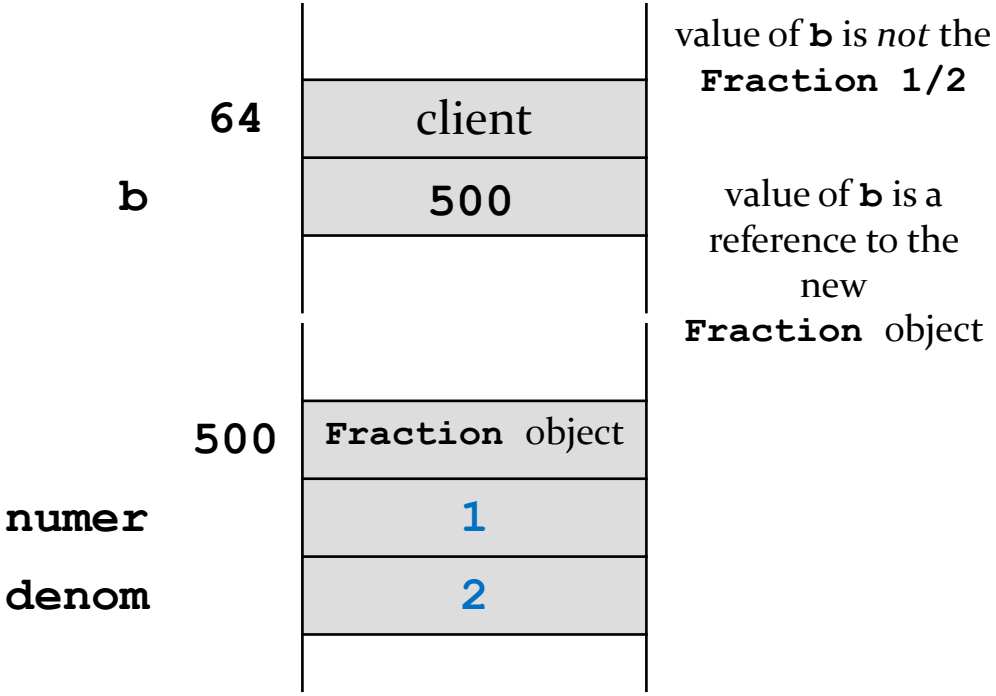
Pass-by-value

- what is the output of the client program?
  - try it and see

- an invoked method runs in its own area of memory that contains storage for its parameters
- each parameter is initialized with *the value* of its corresponding argument

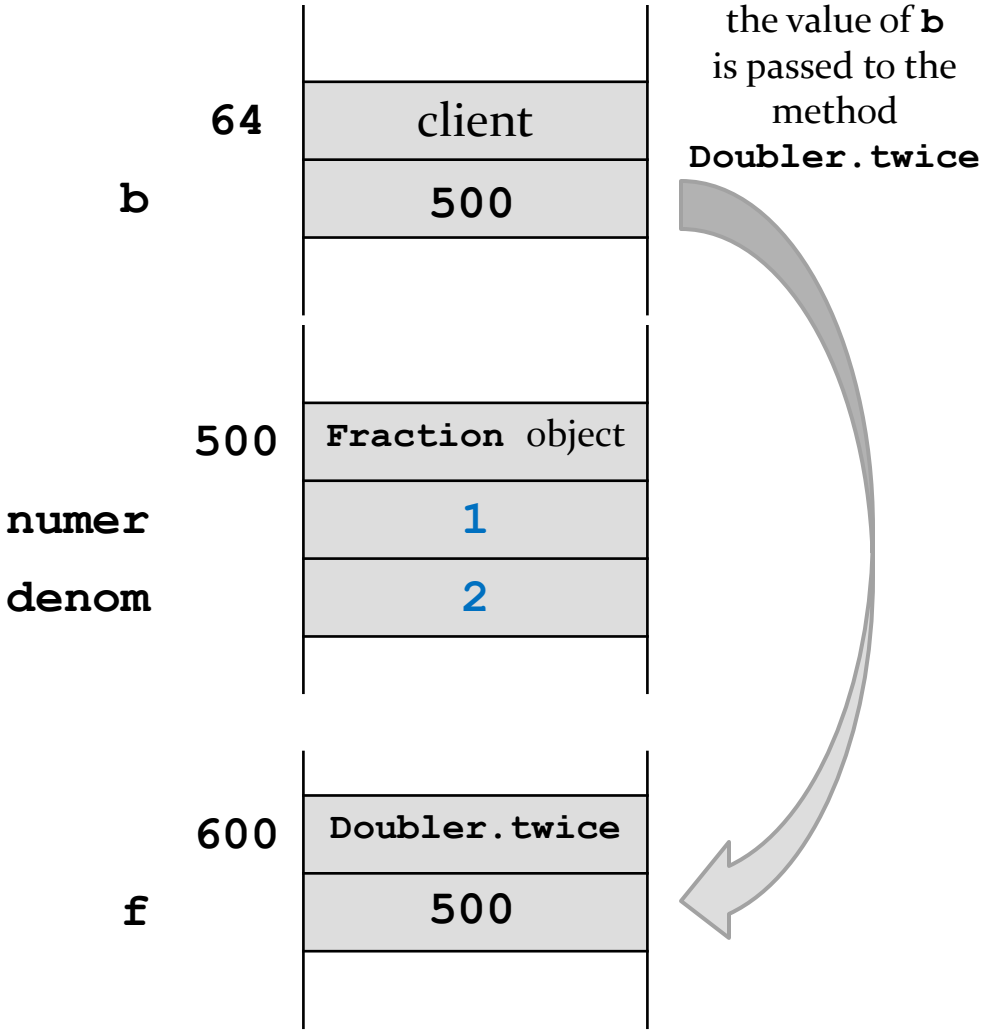
Pass-by-value with Reference Types

```
Fraction b =
  new Fraction(1, 2);
```



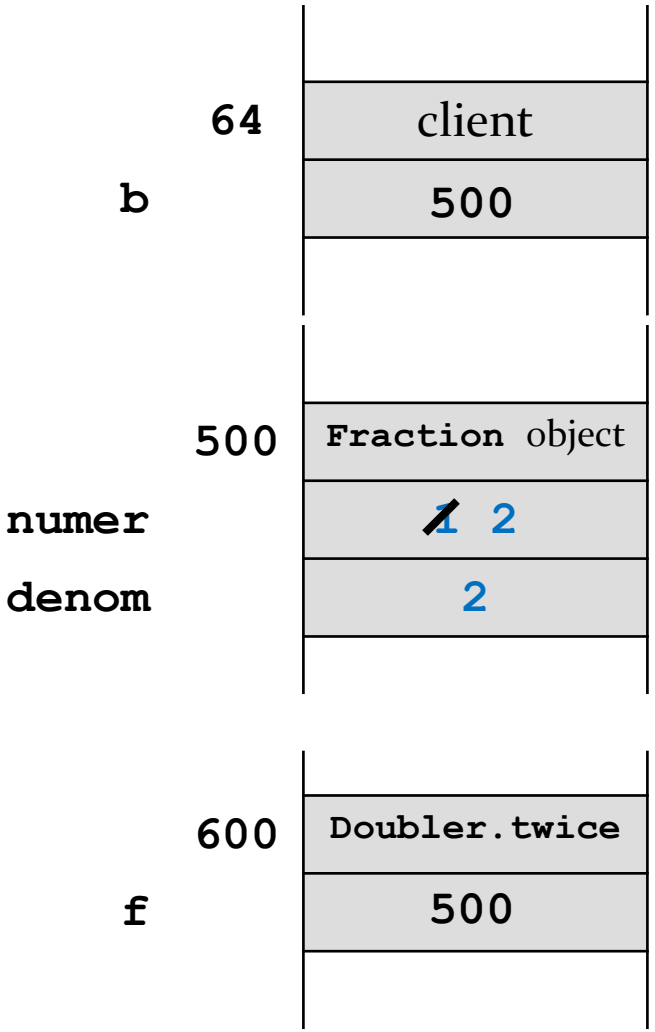
Pass-by-value with Reference Types

```
Fraction b =
  new Fraction(1, 2);
Doubler.twice(b);
```



Pass-by-value with Reference Types

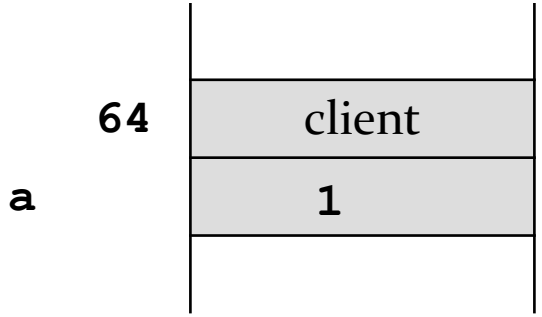
```
Fraction b =
  new Fraction(1, 2);
Doubler.twice(b);
```



Doubler.twice multiplies the numerator of the **Fraction** object by **2**

Pass-by-value with Primitive Types

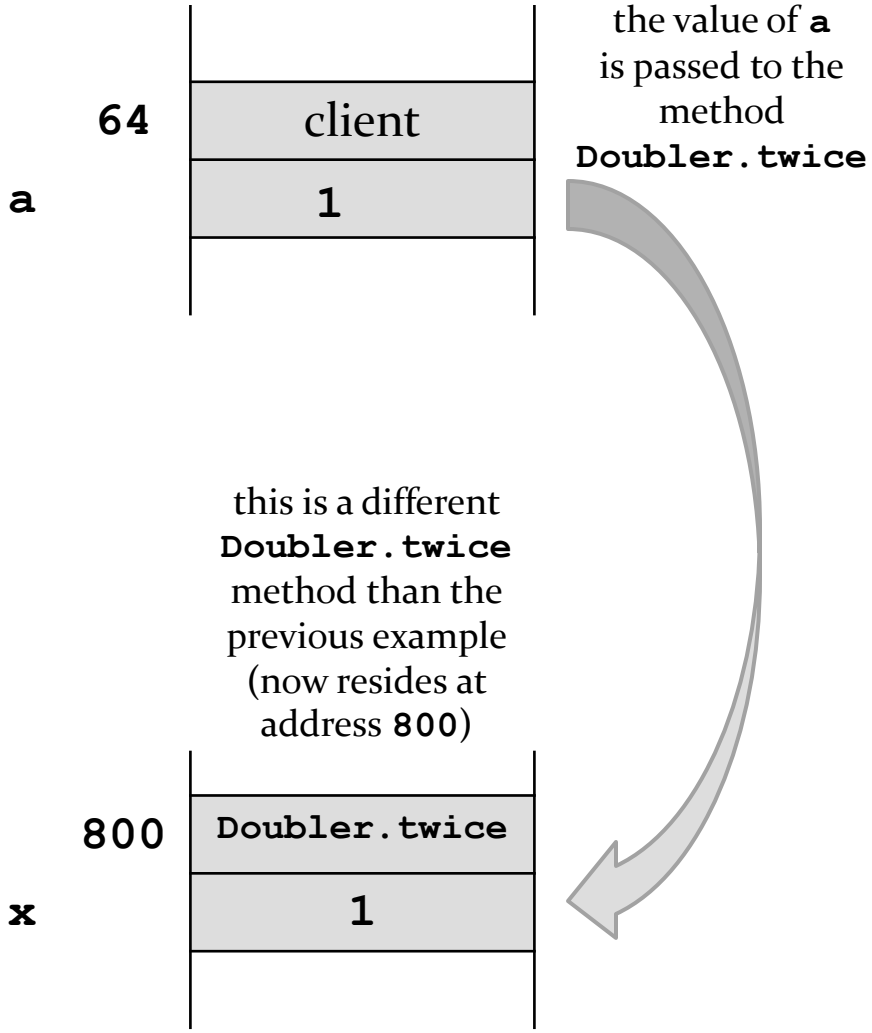
```
int a = 1;
```



value of **a** is the integer value that we stored

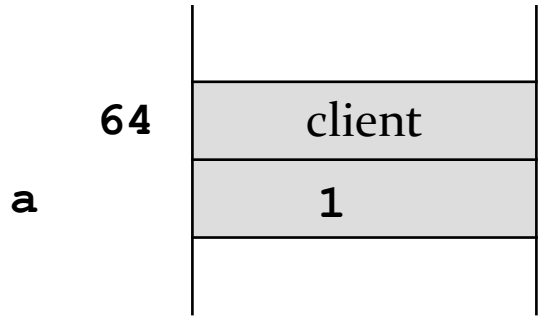
Pass-by-value with Primitive Types

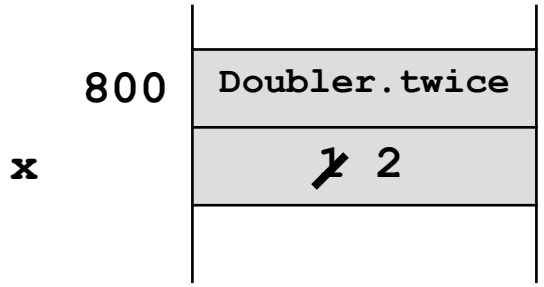
```
int a = 1;
Doubler.twice(a);
```



Pass-by-value with Reference Types

```
int a = 1;
Doubler.twice(a);
```





`Doubler.twice` multiplies the value of `x` by `2`; that's it, nothing else happens

Pass-by-value

- Java uses pass-by-value for *all* types (primitive and reference)
  - an argument of primitive type cannot be changed by a method
  - an argument of reference type can have its state changed by a method

- pass-by-value is used to return a value from a method back to the client