Chapter 5

Methods

Designing Methods

Element	Value in `nap()` example	Required?
Access modifier	`public`	No
Optional specifier	`final`	No
Return type	`void`	Yes
Method name	`nap`	Yes
Parameter list	`(int minutes)`	Yes, but can be empty parentheses
Method signature	`nap(int minutes)`	Yes
Exception list	`throws InterruptedException`	No
Method body	`{ // take a nap }`	Yes, except for `abstract` methods

Access Modifiers

private : within the same class

Package Access : (no keyword !!!) a class in the same package

(or package-private or default access)

protected : a class in the same package or a subclass

public : anywhere

Optional Specifiers

- Can have multiple specifiers in the same method, can specify them in any order
- Access modifiers & optional specifiers must all appear before the return type
- As practice: list the access modifier first

Modifier	Description	Chapter covered
`static`	Indicates the method is a member of the shared class object	Chapter 5
`abstract`	Used in an abstract class or interface when the method body is excluded	Chapter 6
`final`	Specifies that the method may not be overridden in a subclass	Chapter 6
`default`	Used in an interface to provide a default implementation of a method for classes that implement the interface	Chapter 7
`synchronized`	Used with multithreaded code	Chapter 13
`native`	Used when interacting with code written in another language, such as C++	Out of scope
`strictfp`	Used for making floating-point calculations portable	Out of scope

Return Type

- A method must have a return type
- Return type void — can have a return statement with no value, or omit the return statement entirely

```
public class Hike {
  String hike() (int a) {
    if (1 < 2) return "orange";
    return "apple"; // compiler warning
  }
}
```

unreachable code

Although 1 < 2, the compiler won't fully evaluate the if statement & requires a return statement if this condition is false

Method Name

Follow the same rules for variable names

Parameter List

Although the parameter list is required, it doesn't have to contain any parameters

Method Signature

Method Signature = method name + parameter list

types of parameters + their order

```
public class Trip {
  public void visitZoo(String name, int waitTime) {}
  public void visitZoo(String attraction, int rainFall) {}
}
```

can change the order to allow the method to compile

Exception List

```
public class ZooMonorail {
  public void zeroExceptions() {}
  public void oneException()
    throws IllegalArgumentException {}
  public void twoExceptions()
    throws IllegalArgumentException,
           InterruptedException {}
}
```

Method Body

```
public class Bird {
  public void fly1() {}
  public void fly2() // Error
  public void fly3(int a) { int name=5; }
}
```

Declaring Local and Instance Variables

Local Variable Modifiers

1 modifier that can be applied: final (not ~~change value~~)

```
public void zooAnimalCheckup(boolean isWeekend) {
   final int rest;
   if (isWeekend) rest = 5; else rest = 20;

   final var giraffe = new Animal();
   final int[] friends = new int[5];

   rest = 10;
   giraffe = new Animal();
   friends = null;
}
```

```
public void zooAnimalCheckup(boolean isWeekend) {
   final int rest;
   if (isWeekend) rest = 5; // missing else
   System.out.print(rest); // error
}
```

Effectively Final Variables

An effectively final variable is one that is not modified after it is assigned

Instance Variable Modifiers

Modifier	Description	Chapter Covered
`final`	Specifies that the instance variable must be initialized with each instance of the class exactly once	Chapter 5
`volatile`	Instructs the JVM that the value in this variable may be modified by other threads	Chapter 13
`transient`	Used to indicate that an instance variable should not be serialized with the *class*	Chapter 14

```
public class PolarBear {
   final int age = 10;
   final int fishEaten;
   final String name;

   { fishEaten = 10; }

   public PolarBear() {
      name = "Robert";
   }
}
```

→ assign a value in an instance initializer

→ assign a value in a constructor

The compiler does not apply a default value to final var.
A final instance / static variables must receive a value when it is declared or as part of initialization

Working with Varargs

Creating Methods with Varargs

2 RULES

- A method can have at most 1 varargs parameter
- A varargs parameter must be the last parameter in the list

```
public class VisitAttractions {
   public void walk1(int... steps) { }
   public void walk2(int start, int... steps) { }
   public void walk3(int... steps, int start) { } // Error
   public void walk4(int... start, int... steps) { } // Error
}
```

steps is of type **int[]**

Calling Methods with Varargs

2 WAYS
- Pass in an array
- List the elements of the array

```
// Pass an array
int[] data = new int[] {1, 2, 3};
walk1(data);

// Pass a list of data
walk1(1, 2, 3);
```

If we omit the varargs values in the method call, Java will create an array of length zero

Accessing Elements of a Vararg

Just like accessing an array

Using Varargs with Other Method Parameters

```
public class DogWalker {
  public static void walkDog(int start, int... steps) {
    System.out.println(steps.length);
  }

  public static void main(String[] args) {
    walkDog(1); // 0
    walkDog(1, 2); // 1
    walkDog(1, 2, 3); // 2
    walkDog(1, new int[] {4, 5}); // 2
  }}
```

```
walkDog(1, null); // Trigger NullPointerExc. in walkDog()
```

steps is null

Applying Access Modifiers

	private	package	protected	public
the same class	Yes	Yes	Yes	Yes
another class in the same package	No	Yes	Yes	Yes
a subclass in a different package	No	No	Yes	Yes
an unrelated class in a different package	No	No	No	Yes

```
package pond.duck; // Different package than Bird
import pond.goose.Goose;
public class GooseWatcher {
  public void watch() {
    Goose goose = new Goose();
    goose.floatInWater(); // Error
  }
}
```

Note: floatInWater() is a protected method of Bird

```
package pond.swan; // Different package than Bird
import pond.shore.Bird;
public class Swan extends Bird {
  public void swim() {
    floatInWater();
  }
  public void helpOtherSwanSwim() {
    Swan other = new Swan();
    other.floatInWater();
  }
  public void helpOtherBirdSwim() {
    Bird other = new Bird();
    other.floatInWater(); // Error
  }
}
```

Accessing static Data

Accessing a static Variable or Method

```
Snake s = new Snake();
System.out.println(s.hiss);
s = null;
System.out.println(s.hiss);
```

hiss is a static variable of Snake

s is a Snake

hiss is a static variable

> read static variable even s is null

Class vs Instance Membership

A static member **cannot** call an instance member without referencing an instance of the class

```
public class MantaRay {
  private String name = "Sammy";
  public static void first() { }
  public static void second() { }
  public void third() {
    System.out.print(name);
  }
  public static void main( String args[]) {
    first();
    second();
    third(); //Error
  }
}
```

static Variable Modifiers

static variables can be declared with the same modifiers as instance variables

```
public class Panda {
   final static String name = "Ronda";
   static final int bamboo;
   static final double height; //Error
   static { bamboo = 5; }
}
```

final variables must be initialized with a value

static Initializers

All static initializers run when the class is first used, in the order they are defined

```
private static int one;
private static final int two;
private static final int three = 3;
private static final int four; //Error
static {
   one = 1;
   two = 2;
   three = 3; //Error
   two = 4; //Error
}
```

static Imports

- Regular imports: import classes
- static imports: import static members of classes (can use *)

Ex: `import static java.util.Arrays.asList;`

Note: if we create an `asList()` method in our class, Java will give it preference over the imported one, and the method we coded will be used

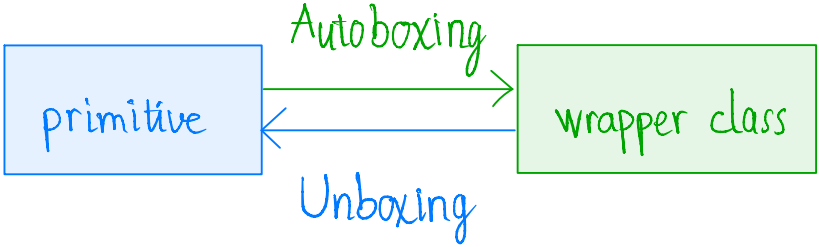
Passing Data among Methods

Passing Objects

```
public class Dog {
  public static void main(String[] args) {
    String name = "Webby";
    speak(name); // name is still "Webby"
  }
  public static void speak(String name) {
    name = "Georgette";
  }
}
```

Java is a "pass-by-value" language: a copy of the variable is made & the method receives that copy

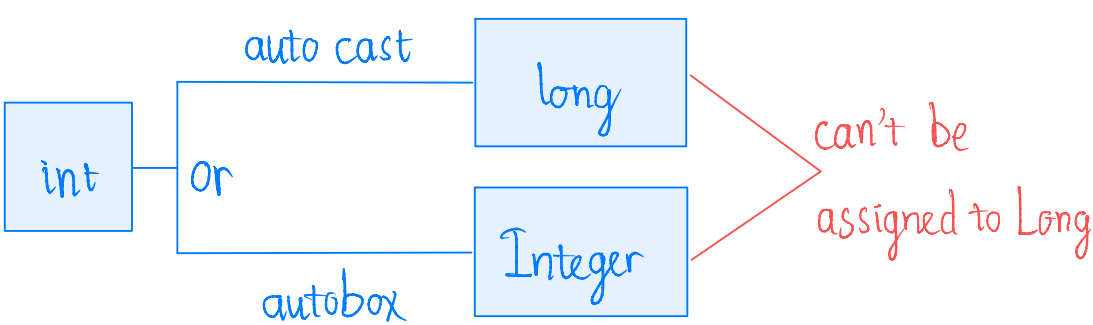
Autoboxing & Unboxing Variable




```
int quack = 5;
Integer quack2 = quack; //Autoboxing
int quack3 = quack2; //Unboxing
```

```
Integer e = Integer.valueOf(9);
long ears = e; // Unboxing, then implicit casting
```

```
Short tail = 8; //OK, 8 is int, narrow int
Long badGorilla = 8; // Error
```



```
Character elephant = null;
char badElephant = elephant; // NullPointerExc.
```

Overloading Methods

Java tries to use the most specific parameter list

```
public class Pelican {
  public void fly ( String s) {
    System.out.println("string");
  }
  public void fly(Object o) {
    System.out.println("object");
  }
  public static void main(String[] args) {
    var p = new Pelican();
    p.fly("test"); p.fly(56); // string - object
  }}
```

```
public class Toucan {
   public void fly(int[] lengths) { }
   public void fly(int... lengths) { }
}
```

Rule	Example of what will be chosen for `glide(1,2)`
Exact match by type	`String glide(int i, int j)`
Larger primitive type	`String glide(long i, long j)`
Autoboxed type	`String glide(Integer i, Integer j)`
Varargs	`String glide(int... nums)`

Review Questions

1/ A E ✓

2/ ~~A~~ B C (use default in interface)

3/ A D ✓

4/ A B C E ✓

5/ A C D ✓

6/ A B F ✓

7/ D F ✓

8/ ~~B~~ D ~~E~~ (lenient: khoan dung)

9/ B D F C (package-access constr.)

10/ B ✓

11/ B E ✓

12/ B ✓

13/ ~~E~~ D (class is not constructed
↳ inst. initializer not run)

14/ E ✓

15/ B ✓

16/ E ✓

17/ B ✓

18/ B D E ✓

19/ B C E ✓

20/ A E ✓ (Integer > Object)

21/ B D ✓

81,0%