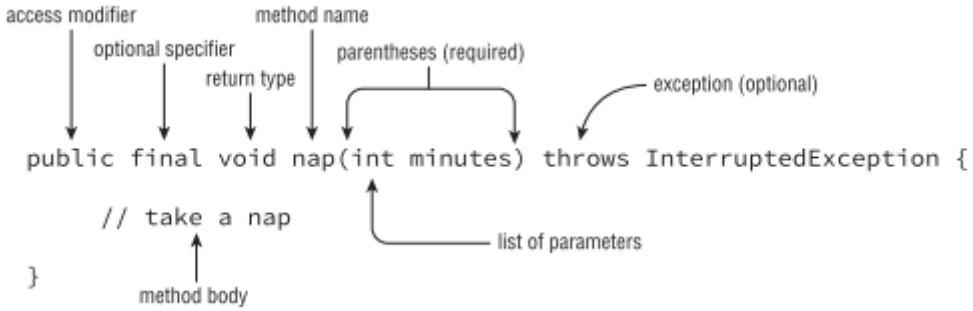
Methods and Encapsulation

1. Designing Methods

Por ejemplo, podemos escribir a método básico: *nap* (siesta):



Este se llama *method declaration* (declaración de método)

La siguiente tabla muestra una referencia sobre la declaración de métodos:

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
Optional exception list	`throws InterruptedException`	No
Method body	`{ // take a nap }`	Yes, but can be empty braces

Access Modifiers
Java ofrece cuatro opciones de modificación de acceso:

1. `public`: Se puede llamar al método desde cualquier clase.
2. `private`: El método solo se puede llamar desde dentro de la misma clase.
3. `protected` El método solo puede invocarse desde clases en el mismo paquete o subclases.
4. Default access (paquete privado) El método solo puede invocarse desde clases en el mismo paquete. Este es complicado porque no hay una palabra clave para el acceso predeterminado. Simplemente omites el modificador de acceso.

Por ejemplo:

```
public void walk1() {}
default void walk2() {} // DOES NOT COMPILE
void public walk3() {} // DOES NOT COMPILE
void walk4() {}
```

1.1. Optional Specifiers

Hay una serie de especificadores opcionales, pero la mayoría de ellos no están en el examen, puede especificarlos en cualquier orden. Y como es opcional, no puedes tener ninguno de ellos. Esto significa que puede tener cero o más especificadores en una declaración de método:

`static`: Usado para métodos de clase.
`abstract`: Se usa cuando no se proporciona un cuerpo de método.
`final`: A nivel de métodos, se usa cuando una subclase no permite que un método sea reemplazado.
`synchronized`: En el OCP pero no en el examen OCA. Se usa para controlar la concurrencia con la técnica de objeto monitor.
`native`: Ni en el OCA u OCP. Se usa cuando se interactúa con código escrito en otro idioma, como C ++.
`strictfp`: Ni en el OCA o OCP. Utilizado para hacer que los cálculos en coma flotante sean portátiles.

Por ejemplo:
```
public void walk1() {}
public final void walk2() {}
public static final void walk3() {}
public final static void walk4() {}
public modifier void walk5() {} // DOES NOT COMPILE
public void final walk6() {} // DOES NOT COMPILE
final public void walk7() {}
```

Del ejemplo anterior, `walk7()` compila porque Java permite que los especificadores opcionales aparezcan antes del modificador de acceso.

1.2. Return Type

El siguiente elemento en una declaración de método es el tipo de devolución. El tipo de devolución puede ser un tipo de Java real, como `String` o `int`. Si no hay un tipo de devolución, se usa la palabra clave `void`. No puede omitir el tipo de devolución

Por ejemplo:
```
public void walk1() { }
public void walk2() { return; }
public String walk3() { return ""; }
public String walk4() { }  // DOES NOT COMPILE
public walk5() { }  // DOES NOT COMPILE
String walk6(int a) { if (a == 4) return ""; }  // DOES NOT COMPILE
```

`walk6()` es un poco complicado. Hay una instrucción de retorno, pero no siempre se consigue ejecutar.

Por ejemplo:
```
int integer() {
  return 9;
}
int longnumber() {]
  return 9L; // DOES NOT COMPILE
}
```

1.3. Method Name

Los nombres de los métodos siguen las mismas reglas que practicamos con nombres de variables. Para revisar, un identificador solo puede contener letras, números, $ o _. Además, el primer carácter no puede ser un número, y las palabras reservadas no están permitidas. Por convención, los métodos comienzan con una letra minúscula pero no están obligados a hacerlo. Por ejemplo:

```
public void walk1() { }
public void 2walk() { } // DOES NOT COMPILE
public walk3 void() { } // DOES NOT COMPILE
public void Walk_$() { }
public void() { } // DOES NOT COMPILE
```

1.4. Parameter List

Aunque la lista de parámetros es obligatoria, no tiene que contener ningún parámetro. Esto significa que puede tener un par de paréntesis vacíos después del nombre del método. Por ejemplo:

```
public void walk1() { }
public void walk2 { } // DOES NOT COMPILE
public void walk3(int a) { }
public void walk4(int a; int b) { }  // DOES NOT COMPILE
public void walk5(int a, int b) { }
```

1.5. Optional Exception List

En Java, el código puede indicar que algo salió mal lanzando una excepción. Esto lo cubrirá más adelante, por ahora, solo necesita saber que es opcional y en qué parte de la firma del método va si está presente. Por ejemplo:

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

1.6. Method Body

La parte final de una declaración de método es el cuerpo del método (excepto los métodos e interfaces abstractos. Un cuerpo de método es simplemente un bloque de código. Tiene llaves que contienen cero o más sentencias de Java. Por ejemplo:

```
public void walk1() { }
public void walk2; // DOES NOT COMPILE
public void walk3(int a) { int name = 5; }
```

2. Working with Varargs

Un parámetro `vararg` debe ser el último elemento en la lista de parámetros de un método. Esto implica que solo se le permite tener un parámetro `vararg` por método. Por ejemplo:

```
public void walk1(int... nums) { }
public void walk2(int start, int... nums) { }
public void walk3(int... nums, int start) { } // DOES NOT COMPILE
public void walk4(int... start, int... nums) { } // DOES NOT COMPILE
```

Cuando llama a un método con un parámetro `vararg`, tiene una opción. Puede pasar una matriz, o puede listar los elementos de la matriz y dejar que Java la cree por usted. Incluso puede omitir los valores `vararg` en la llamada al método y Java creará una matriz de longitud cero.

¿Puedes averiguar por qué cada llamada a método genera lo siguiente?

```
15: public static void walk(int start, int... nums) {
16:  System.out.println(nums.length);
17: }
18: public static void main(String[] args) {
19:  walk(1);                    // 0
20:  walk(1, 2);                 // 1
21:  walk(1, 2, 3);                 // 2
22:  walk(1, new int[] {4, 5});     // 2
23: }
```

Todavía es posible pasar `null` explícitamente:

```
walk(1, null);      // throws a NullPointerException
```

Como `null` no es un `int`, Java lo trata como una referencia de matriz que pasa a ser `null`.

Acceder a un parámetro `vararg` también es como acceder a una matriz. Utiliza indexación de matriz. Por ejemplo:

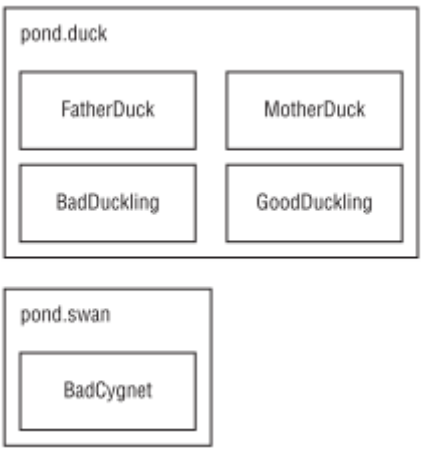
```
16: public static void run(int... nums) {
17:    System.out.println(nums[1]);
18: }
19: public static void main(String[] args) {
20:    run(11, 22);      // 22
21: }
```

3. Applying Access Modifiers

Vamos a discutirlos en orden de lo más restrictivo a lo menos restrictivo:
`private`: solo accesible dentro de la misma clase
`default access` (paquete privado): clases privadas y otras clases en el mismo paquete
`protected`: paquete privado y clases hijas.
`public`: protegido y clases en los otros paquetes

3.1. Private Access

El acceso privado es fácil. Solo el código en la misma clase puede llamar a métodos privados o acceder a campos privados. Por ejemplo:



Este es un código perfectamente legal porque todo es una clase:

```
1: package pond.duck;
2: public class FatherDuck {
3:  private String noise = "quack";
4:  private void quack() {
5:    System.out.println(noise);      // private access is ok
6:  }
7:  private void makeNoise() {
8:    quack();                        // private access is ok
9:  } }
```

Ahora agregamos otra clase:

```
1: package pond.duck;
2: public class BadDuckling {
3:  public void makeNoise() {
```

```
4:    FatherDuck duck = new FatherDuck();
5:    duck.quack();                          // DOES NOT COMPILE
6:    System.out.println(duck.noise);        // DOES NOT COMPILE
7:  } }
```

3.2. Default (Package Private) Access

Cuando no hay ningún modificador de acceso, Java usa el valor predeterminado, que es el acceso privado del paquete. Esto significa que el miembro es "privado" para las clases en el mismo paquete. En otras palabras, solo las clases del paquete pueden acceder a él. Por ejemplo:

```
package pond.duck;
public class MotherDuck {
  String noise = "quack";
  void quack() {
    System.out.println(noise);      // default access is ok
  }
  private void makeNoise() {
    quack();                     // default access is ok
  } }
```

Continuamos con otra clase:

```
package pond.duck;
public class GoodDuckling {
  public void makeNoise() {
    MotherDuck duck = new MotherDuck();
    duck.quack();                          // default access
    System.out.println(duck.noise);           // default access
  } }
```

Finalmente, analice:

```
package pond.swan;
import pond.duck.MotherDuck;          // import another package
public class BadCygnet {
  public void makeNoise() {
    MotherDuck duck = new MotherDuck();
    duck.quack();                        // DOES NOT COMPILE
    System.out.println(duck.noise);      // DOES NOT COMPILE
  } }
```

3.3. Protected Access

El modificador de acceso protegido agrega la capacidad de acceder a los miembros de una clase principal. Por ejemplo:

Primero, creamos una clase `Bird` y otorgamos acceso protegido a sus miembros:

```
package pond.shore;
public class Bird {
  protected String text = "floating";          // protected access
  protected void floatInWater() {               // protected access
    System.out.println(text);
  } }
```

A continuación, creamos una subclase:

```
package pond.goose;
import pond.shore.Bird;               // in a different package
public class Gosling extends Bird {     // extends means create subclass
  public void swim() {
    floatInWater();               // calling protected member
```

```
    System.out.println(text);     // calling protected member
  } }
```

La definición de protegido permite el acceso a subclases y clases en el mismo paquete. Ahora intentemos lo mismo con un paquete diferente:

```
package pond.inland;
import pond.shore.Bird;               // different package than Bird
public class BirdWatcherFromAfar {
  public void watchBird() {
    Bird bird = new Bird();
    bird.floatInWater();               // DOES NOT COMPILE
    System.out.println(bird.text);     // DOES NOT COMPILE
  } }
```

Considera la siguiente clase:

```
1:  package pond.swan;
2:  import pond.shore.Bird;     // in different package than Bird
3:  public class Swan extends Bird {     // but subclass of bird
4:    public void swim() {
5:      floatInWater();               // package access to superclass
6:      System.out.println(text);     // package access to superclass
7:    }
8:    public void helpOtherSwanSwim() {
9:      Swan other = new Swan();
10:     other.floatInWater();          // package access to superclass
11:     System.out.println(other.text);// package access to superclass
12:   }
13:   public void helpOtherBirdSwim() {
14:     Bird other = new Bird();
15:     other.floatInWater();               // DOES NOT COMPILE
16:     System.out.println(other.text);      // DOES NOT COMPILE
17:   }
18: }
```

Las reglas protegidas se aplican bajo dos escenarios:

1. Un miembro se usa sin referirse a una variable. En este caso, estamos aprovechando la herencia y se permite el acceso protegido.
2. Un miembro se usa a través de una variable. En este caso, las reglas para el tipo de referencia de la variable son importantes.

3.4. Public Access

público significa que cualquiera puede acceder al miembro desde cualquier lugar. Por ejemplo:

```
package pond.duck;
public class DuckTeacher {
  public String name = "helpful";     // public access
  public void swim() {                // public access
    System.out.println("swim");
  } }
```

`DuckTeacher` permite el acceso a cualquier clase que lo desee. Ahora podemos probarlo:

```
package pond.goose;
import pond.duck.DuckTeacher;
public class LostDuckling {
  public void swim() {
    DuckTeacher teacher = new DuckTeacher();
    teacher.swim();                                  // allowed
```

```
    System.out.println("Thanks" + teacher.name);      // allowed
  } }
```

Para revisar los modificadores de acceso, asegúrese de saber por qué todo en la siguiente tabla:

¿Puede acceder?	¿Si el miembro es privado?	¿Si el miembro tiene default access?	¿Si el miembro es protected?	¿Si el miembro es public?
Miembros en la misma clase	SI	SI	SI	SI
Miembro en otra clase en el mismo paquete	No	SI	SI	SI
Miembro en una superclase en un diferente paquete	No	No	SI	SI
Método/Atributo en una no superclase (instancia) en diferente paquete	No	No	No	SI

3.5. Designing Static Methods and Fields

Los métodos estáticos no requieren una instancia de la clase. Se comparten entre todos los usuarios de la clase. Puedes pensar en estática como un miembro del objeto de una sola clase que existe independientemente de cualquier instancia de esa clase.

¿Cada clase tiene su propia copia del código?
Cada clase tiene una copia de las variables de instancia. Solo hay una copia del código para los métodos de instancia. Cada instancia de la clase puede llamarlo tantas veces como quiera. Sin embargo, cada llamada de un método de instancia (o cualquier método) obtiene espacio en la pila para parámetros de método y variables locales.
Lo mismo ocurre con los métodos estáticos. Hay una copia del código. Los parámetros y las variables locales van en la pila.

Por ejemplo:

```
public class Koala {
  public static int count = 0;                // static variable
  public static void main(String[] args) {       // static method
    System.out.println(count);
  }
}
```

3.6. Calling a Static Variable or Method

Por ejemplo:

```
System.out.println(Koala.count);
Koala.main(new String[0]);
```

Este código es perfectamente legal:

```
5: Koala k = new Koala();
6: System.out.println(k.count);          // k is a Koala
7: k = null;
8: System.out.println(k.count);          // k is still a Koala
```

3.7. Static vs. Instance

Un miembro estático no puede llamar a un miembro de instancia. Por ejemplo, analicemos:

```
1:  public class Gorilla {
2:    public static int count;
3:    public static void addGorilla() { count++; }
4:    public void babyGorilla() { count++; }
5:    public void announceBabies() {
6:      addGorilla();
7:      babyGorilla();
8:    }
9:    public static void announceBabiesToEveryone() {
10:     addGorilla();
11:     babyGorilla();     // DOES NOT COMPILE
12:   }
13:   public int total;
14:   public static average = total / count;  // DOES NOT COMPILE
15: }
```

Type	Calling	Legal?	How?
Static method	Another static method or variable	Yes	Using the classname
Static method	An instance method or variable	No	
Instance method	A static method or variable	Yes	Using the classname or a reference variable
Instance method	Another instance method or variable	Yes	Using a reference variable

3.8. Static Variables

Algunas variables estáticas deben cambiar a medida que se ejecuta el programa. Otras variables estáticas están destinadas a nunca cambiar durante el programa. Este tipo de variable se conoce como una constante. Usan todas las letras mayúsculas con guiones bajos entre "palabras". Por ejemplo:

```
// Esta variable static es mutable
public class Initializers {
  private static int counter = 0;          // initialization
}

// La siguiente variable es inmutable
public class Initializers {
  private static final int NUM_BUCKETS = 45;
  public static void main(String[] args) {
     NUM_BUCKETS = 5;  // DOES NOT COMPILE
  } }
```

¿Crees que lo siguiente compila?

```
private static final ArrayList<String> values = new ArrayList<>();
public static void main(String[] args) {
  values.add("changed");
}
```

Realmente compila. `values` es una variable de referencia. Todo lo que el compilador puede hacer es verificar que no intentemos reasignar final `values` para apuntar a un objeto diferente.

3.9. Static Initialization

Agregan la palabra clave `static` para especificar que se deben ejecutar cuando se usa por primera vez la clase. Por ejemplo:

```
private static final int NUM_SECONDS_PER_HOUR;
static {
  int numSecondsPerMinute = 60;
  int numMinutesPerHour = 60;
  NUM_SECONDS_PER_HOUR = numSecondsPerMinute * numMinutesPerHour;
}
```

El inicializador estático se ejecuta cuando la clase se utiliza por primera vez. La clave aquí es que el inicializador estático es la primera asignación. Y dado que ocurre desde el principio, está bien. Probemos otro ejemplo:

```
14: private static int one;
15: private static final int two;
16: private static final int three = 3;
17: private static final int four;     // DOES NOT COMPILE
18: static {
19:   one = 1;
20:   two = 2;
21:   three = 3;     // DOES NOT COMPILE
22:   two = 4;     // DOES NOT COMPILE
23: }
```

3.10. Static Imports

Las importaciones regulares son para importar clases. Las importaciones estáticas son para importar miembros estáticos de clases. Al igual que las importaciones regulares, puede usar un comodín o importar un miembro específico. La idea es que no debas especificar de dónde proviene cada método o variable estática cada vez que la utilizas. Por ejemplo:

```
import java.util.List;
import static java.util.Arrays.asList;          // static import
public class StaticImports {
  public static void main(String[] args) {
    List<String> list = asList("one", "two");     // no Arrays.
  } }
```

¿Puedes descubrir qué está mal con cada uno?

```
1: import static java.util.Arrays; // DOES NOT COMPILE
2: import static java.util.Arrays.asList;
3: static import java.util.Arrays.*;  // DOES NOT COMPILE
4: public class BadStaticImports {
5:   public static void main(String[] args) {
6:    Arrays.asList("one");  // DOES NOT COMPILE
7:  } }
```

El compilador se quejará si intenta realizar explícitamente una importación estática de dos métodos con el mismo nombre o dos variables estáticas con el mismo nombre. Por ejemplo:

```
import static statics.A.TYPE;
import static statics.B.TYPE;     // DOES NOT COMPILE
```

4. Passing Data Among Methods

Java es un lenguaje de "paso por valor". Esto significa que se realiza una copia de la variable y el método recibe esa copia. Las asignaciones realizadas en el método no afectan a quien lo llama. Veamos un ejemplo:

```
2: public static void main(String[] args) {
3:   int num = 4;
4:   newNumber(5);
5:   System.out.println(num);     // 4
6: }
7: public static void newNumber(int num) {
8:   num = 8;
9: }
```

Como ejemplo, tenemos un código que llama a un método en `StringBuilder` pasado al método:

```
public static void main(String[] args) {
  StringBuilder name = new StringBuilder();
  speak(name);
  System.out.println(name); // Webby
}
public static void speak(StringBuilder s) {
  s.append("Webby");
}
```

En este caso, la salida es `Webby` porque el método simplemente llama a un método en el parámetro. No reasigna el nombre a un objeto diferente.

Probemos un ejemplo. Preste atención a los tipos de devolución:

```
1: public class ReturningValues {
2:  public static void main(String[] args) {
3:    int number = 1;                          // 1
4:    String letters = "abc";                  // abc
5:    number(number);                          // 1
6:    letters = letters(letters);              // abcd
7:    System.out.println(number + letters);    // 1abcd
8:  }
9:  public static int number(int number) {
10:   number++;
11:   return number;
12:  }
13:  public static String letters(String letters) {
14:   letters += "d";
15:   return letters;
16:  }
17: }
```

5. Overloading Methods

La sobrecarga de métodos ocurre cuando hay diferentes firmas de métodos con el mismo nombre, pero diferentes parámetros de tipo o permite diferentes números. Por ejemplo:

```
public void fly(int numMiles) { }
public void fly(short numFeet) { }
public boolean fly() { return false; }
void fly(int numMiles, short numFeet) { }
public void fly(short numFeet, int numMiles) throws Exception { }
```

Podemos tener un tipo diferente, más tipos o los mismos tipos en un orden diferente. También tenga en cuenta que el modificador de acceso y la lista de excepciones son irrelevantes para la sobrecarga. Por ejemplo:

```
public void fly(int numMiles) { }
public int fly(int numMiles) { }     // DOES NOT COMPILE

public void fly(int numMiles) { }
public static void fly(int numMiles) { }     // DOES NOT COMPILE
```

5.1. Overloading and Varargs

Which method do you think is called if we pass an `int[]`?

```
public void fly(int[] lengths) { }
public void fly(int... lengths) { }     // DOES NOT COMPILE
```

¡Pregunta capciosa! Recuerde que Java trata `varargs` como si fueran una matriz. Esto significa que la firma del método es la misma para ambos métodos. Como no tenemos permitido sobrecargar los métodos con la misma lista de parámetros, este código no se compila.

5.2. Autoboxing

¿Qué pasa si tenenos versions con primitive y entero?

```
public void fly(int numMiles) { }
public void fly(Integer numMiles) { }
```

Java intenta usar la lista de parámetros más específica que puede encontrar. Cuando la versión `int` primitiva no está presente, se realiza un *autobox*.

5.3. Reference Types

¿Qué pasa si tenemos una versión tanto primitiva como entera?

```
public class ReferenceTypes {
  public void fly(String s) {
    System.out.print("string ");
  }
  public void fly(Object o) {
    System.out.print("object ");
  }
  public static void main(String[] args) {
    ReferenceTypes r = new ReferenceTypes();
    r.fly("test");
    r.fly(56);
  } }
```

La respuesta es "string object". La primera llamada es una Cadena y encuentra una coincidencia directa. No hay ninguna razón para usar la versión de Objeto cuando hay una lista de parámetros de Cadena que está esperando ser llamada. La segunda llamada busca una lista de parámetros int. Cuando no encuentra uno, se realiza un *autobox* a `Integer`.

5.4. Primitives

¿Qué crees que pasa aquí?

```
public class Plane {
  public void fly(int i) {
    System.out.print("int ");
```

```
  }
  public void fly(long l) {
    System.out.print("long ");
  }
  public static void main(String[] args) {
     Plane p = new Plane();
     p.fly(123);
     p.fly(123L);
  } }
```

La respuesta es `int long`. Tenga en cuenta que Java solo puede aceptar tipos más amplios. Un int se puede pasar a un método que toma un parámetro largo. Java no se convertirá automáticamente a un tipo más estrecho.

5.5. Putting It All Together

El orden oficial:

Rule	**Ejemplo de qué se escoge en** `glide(1,2)`
Coincidencia exacta por tipo	`public String glide(int i, int j) {}`
Tipo primitivo de mayor capacidad	`public String glide(long i, long j) {}`
Tipo *Autobox*	`public String glide(Integer i, Integer j) {}`
`Varargs`	`public String glide(int... nums) {}`

¿Qué crees que esto produce?

```
public class Glider2 {
  public static String glide(String s) {
    return "1";
  }
  public static String glide(String... s) {
    return "2";
  }
  public static String glide(Object o) {
    return "3";
  }
  public static String glide(String s, String t) {
    return "4";
  }
  public static void main(String[] args) {
    System.out.print(glide("a"));
    System.out.print(glide("a", "b"));
    System.out.print(glide("a", "b", "c"));
  } }
```

Imprime `142`. Otro ejemplo:

```
public class TooManyConversions {
  public static void play(Long l) { }
  public static void play(Long... l) { }
  public static void main(String[] args) {
    play(4);     // DOES NOT COMPILE
    play(4L);     // calls the Long version
  } }
```

6. Creating Constructors

Aquí hay un ejemplo:

```
public class Bunny {
  public Bunny() {
    System.out.println("constructor");
  }
}

new Bunny()
```

Cuando Java ve la palabra clave nueva, asigna memoria para el nuevo objeto. Java también busca un constructor y lo llama. Un constructor se usa generalmente para inicializar variables de instancia. La palabra clave `this` le dice a Java que desea hacer referencia a una variable de instancia. La mayoría de las veces, esto es opcional.

Por ejemplo:

```
1: public class Bunny {
2:   private String color;
3:   private int height;
4:   private int length;
5:   public Bunny(int length, int theHeight) {
6:     length = this.length;     // backwards - no good!
7:     height = theHeight;          // fine because a different name
8:     this.color = "white";     // fine, but redundant
9:   }
10: public static void main(String[] args) {
11:  Bunny b = new Bunny(1, 2);
12:  System.out.println(b.length + " " + b.height + " " + b.color);
13:  } }
```

6.1. Default Constructor

Este constructor creado por Java se llama el constructor predeterminado. A veces lo llamamos el constructor de no argumentos por defecto para mayor claridad. Aquí hay un ejemplo:

```
public class Rabbit {
  public static void main(String[] args) {
    Rabbit rabbit = new Rabbit();          // Calls default constructor
  }
}
```

¿Cuál de estas clases cree que tiene un constructor predeterminado?

```
class Rabbit1 {
}
class Rabbit2 {
  public Rabbit2() { }
}
class Rabbit3 {
  public Rabbit3(boolean b) { }
}
class Rabbit4 {
  private Rabbit4() { }
}
```

Solo Rabbit1 obtiene un constructor predeterminado sin argumento. Echemos un vistazo rápido a cómo llamar a estos constructores:

```
1: public class RabbitsMultiply {
2:   public static void main(String[] args) {
3:    Rabbit1 r1 = new Rabbit1();
4:    Rabbit2 r2 = new Rabbit2();
5:    Rabbit3 r3 = new Rabbit3(true);
6:    Rabbit4 r4 = new Rabbit4(); // DOES NOT COMPILE
7:   } }
```

6.2. Overloading Constructors

Puede tener varios constructores en la misma clase, siempre que tengan diferentes firmas de métodos. Los constructores deben tener diferentes parámetros para estar sobrecargados. Este ejemplo muestra dos constructores:

```
public class Hamster {
  private String color;
  private int weight;
  public Hamster(int weight) {                // first constructor
    this.weight = weight;
    color = "brown";
  }
  public Hamster(int weight, String color) {     // second constructor
   this.weight = weight;
   this.color = color;
  }
}
```

Sin embargo, hay un problema aquí. Hay un poco de duplicación. Lo que realmente queremos es que el primer constructor llame al segundo constructor con dos parámetros. Es posible que tengas la tentación de escribir esto:

```
public Hamster(int weight) {
  Hamster(weight, "brown");     // DOES NOT COMPILE
}
```

O también:

```
public Hamster(int weight) {
  new Hamster(weight, "brown");     // Compiles but does not do what we want
}
```

Java proporciona una solución: `this`

```
public Hamster(int weight) {
  this(weight, "brown");
}
```

`this()` tiene una regla especial que necesitas saber. Si elige llamarlo, la llamada a `this()` debe ser la primera declaración no comentada en el constructor:

```
3: public Hamster(int weight) {
4:   System.out.println("in constructor");
5:   // ready to call this
6:   this(weight, "brown");     // DOES NOT COMPILE
7: }
```

6.3. Final Fields

Las variables de instancia final deben tener asignado un valor exactamente una vez. Vimos que esto sucedía en la línea de la declaración y en un inicializador de instancia. Hay una ubicación más que se puede hacer esta asignación: en el constructor.

```
public class MouseHouse {
  private final int volume;
  private final String name = "The Mouse House";
  public MouseHouse(int length, int width, int height) {
   volume = length * width * height;
  }}
```

El constructor es parte del proceso de inicialización, por lo que está permitido asignarle variables de instancia finales.

6.4. Order of Initialization

Necesitas saber este orden de memoria:

1. Si hay una superclase, inicialícela primero (cubriremos esta regla más adelante. Por ahora, simplemente diga "sin superclase" y continúe con la siguiente regla).
2. Declaraciones de variables estáticas e inicializadores estáticos en el orden en que aparecen en el archivo.
3. Declaraciones de variables de instancia e inicializadores de instancia en el orden en que aparecen en el archivo.
4. El constructor

Probemos el primer ejemplo, analicémoslo:

```
1: public class InitializationOrderSimple {
2:   private String name = "Torchie";
3:   { System.out.println(name); }
4:   private static int COUNT = 0;
5:   static { System.out.println(COUNT); }
6:   static { COUNT += 10; System.out.println(COUNT); }
7:   public InitializationOrderSimple() {
8:     System.out.println("constructor");
9:   } }
```

```
1: public class CallInitializationOrderSimple {
2:   public static void main(String[] args) {
3:     InitializationOrderSimple init = new InitializationOrderSimple();
4:   } }
```

La salida es:

```
0
10
Torchie
constructor
```

Otro ejemplo:

```
1: public class YetMoreInitializationOrder {
2:   static { add(2); }
3:   static void add(int num) { System.out.print(num + " "); }
4:   YetMoreInitializationOrder() { add(5); }
5:   static { add(4); }
6:   { add(6); }
7:   static { new YetMoreInitializationOrder(); }
8:   { add(8); }
```

```
9:   public static void main(String[] args) { } }
```

La respuesta correcta es `2 4 6 8 5`.

7. Encapsulating Data

La encapsulación significa que configuramos la clase, por lo que solo los métodos en la clase con las variables pueden referirse a las variables de instancia. Las personas que llaman deben usar estos métodos. Tomemos un ejemplo:

```
1: public class Swan {
2:  private int numberEggs;                    // private
3:  public int getNumberEggs() {              // getter
4:    return numberEggs;
5:  }
6:  public void setNumberEggs(int numberEggs) {   // setter
7:    if (numberEggs >= 0)                        // guard condition
8:      this.numberEggs = numberEggs;
9:  } }
```

Para la encapsulación, recuerde que los datos (una variable de instancia) son privados y los `getters`/`setters` son públicos. Java define una convención de nomenclatura que se usa en JavaBeans. Los JavaBeans son componentes de software reutilizables. JavaBeans llama a una variable de instancia una propiedad. Lo único que necesita saber sobre JavaBeans para el examen son las convenciones de nomenclatura que figuran en la siguiente tabla:

Regla	**Ejemplo**
Propiedades son privadas.	`private int numEggs;` `private boolean happy;`

Los métodos de `Getter` comienzan con `is` la propiedad es un booleano.	`public boolean isHappy() {`
	`  return happy;`
	`}`

Los métodos `Getter` comienzan con `get` si la propiedad no es booleana.	`public int getNumEggs() {`
	`  return numEggs;`
	`}`

Los métodos Setter comienzan con set.	`public void setHappy(boolean happy) {`
	`  this.happy = happy;`
	`}`

El nombre del método debe tener un prefijo de `set/get/is`, seguido de la primera letra de la propiedad en mayúscula, seguido del resto del nombre de la propiedad.	`public void setNumEggs(int num) {`
	`  numEggs = num;`
	`}`

Vea si puede averiguar qué líneas siguen las convenciones de nombres de JavaBeans:

```
12: private boolean playing;
13: private String name;
14: public boolean getPlaying() { return playing; }
15: public boolean isPlaying() { return playing; }
16: public String name() { return name; }
17: public void updateName(String n) { name = n; }
18: public void setname(String n) { name = n; }
//Sólo 12, 13 y 15
```

7.1. Creating Immutable Classes

Las clases inmutables son útiles porque sabes que siempre serán las mismas. Puede pasarlos por su aplicación con la garantía de que la persona que llama no cambió nada.

Un paso para hacer que una clase sea inmutable es omitir a los establecedores. Pero espere: todavía queremos que la persona que llama pueda especificar el valor inicial. Constructores al rescate:

```
public class ImmutableSwan {
  private int numberEggs;
  public ImmutableSwan(int numberEggs) {
    this.numberEggs = numberEggs;
  }
  public int getNumberEggs() {
    return numberEggs;
  } }
```

Para revisar, la encapsulación se refiere a evitar que las personas que llaman cambien las variables de instancia directamente. La inmutabilidad se refiere a evitar que las personas que llaman modifiquen las variables de instancia.

8. Writing Simple Lambdas

En Java 8, el lenguaje agregó la capacidad de escribir código usando otro estilo. La programación funcional es una forma de escribir código más declarativamente. Usted especifica lo que quiere hacer en lugar de tratar con el estado de los objetos. Te enfocas más en las expresiones que en los bucles.

La programación funcional usa expresiones lambda para escribir código. Una expresión lambda es un bloque de código que se pasa. Puedes pensar en una expresión lambda como método anónimo. Tiene parámetros y un cuerpo como lo hacen los métodos completos, pero no tiene un nombre como un método real. Solo las expresiones lambda más simples están en el examen OCA.

8.1. Lambda Example

Nuestro objetivo es imprimir todos los animales en una lista de acuerdo con algunos criterios:

```
public class Animal {
  private String species;
  private boolean canHop;
  private boolean canSwim;
  public Animal(String speciesName, boolean hopper, boolean swimmer) {
    species = speciesName;
    canHop = hopper;
    canSwim = swimmer;
  }
   public boolean canHop() { return canHop; }
   public boolean canSwim() { return canSwim; }
   public String toString() { return species; }
}

public interface CheckTrait {
  boolean test(Animal a);
}

public class CheckIfHopper implements CheckTrait {
  public boolean test(Animal a) {
    return a.canHop();
  }
}
```

Ahora tenemos todo lo que necesitamos para escribir nuestro código para encontrar los animales que brincan (*hop*):

```
1: public class TraditionalSearch {
2:   public static void main(String[] args) {
3:     List<Animal> animals = new ArrayList<Animal>();  // list of animals
4:     animals.add(new Animal("fish", false, true));
5:     animals.add(new Animal("kangaroo", true, false));
6:     animals.add(new Animal("rabbit", true, false));
7:     animals.add(new Animal("turtle", false, true));
8:
9:     print(animals, new CheckIfHopper());      // pass class that does check
10:   }
11:  private static void print(List<Animal> animals, CheckTrait checker) {
12:     for (Animal animal : animals) {
13:       if (checker.test(animal))               // the general check
14:         System.out.print(animal + " ");
15:     }
16:     System.out.println();
17:   }
18: }
```

Podríamos reemplazar la línea 9 por la siguiente, que usa una lambda:

```
9:    print(animals, a -> a.canHop());
```

Solo tenemos que agregar una línea de código, no hay necesidad de una clase adicional para hacer algo simple. Aquí está esa otra línea:

```
print(animals, a -> a.canSwim());
```

¿Qué hay de los animales que no pueden nadar (*swim*)?

```
print(animals, a -> ! a.canSwim());
```

Este código usa un concepto llamado ejecución diferida. La ejecución diferida significa que el código está especificado ahora, pero se ejecutará más tarde. En este caso, más tarde es cuando el método `print()` lo llama.

8.2. Lambda Syntax

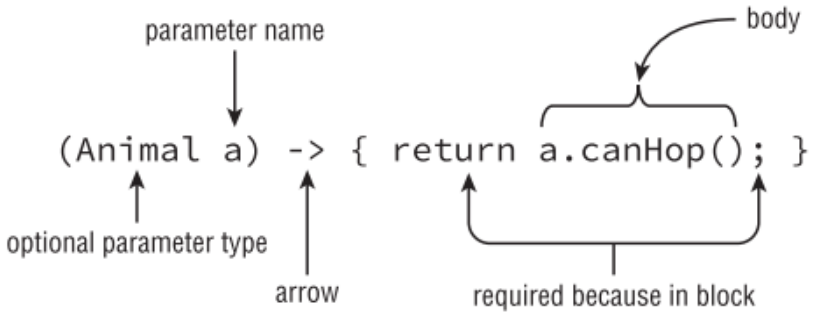
La sintaxis de lambdas es complicada porque muchas partes son opcionales. Estas dos líneas hacen exactamente lo mismo:

```
(Animal a) -> { return a.canHop(); }     //(1)
a -> a.canHop()                          //(2)
```

Para (1): estamos pasando este lambda como el segundo parámetro del método `print()`. Ese método espera un `CheckTrait` como el segundo parámetro. Como en su lugar aprobamos una lambda, Java intenta asignar nuestra lambda a esa interfaz:

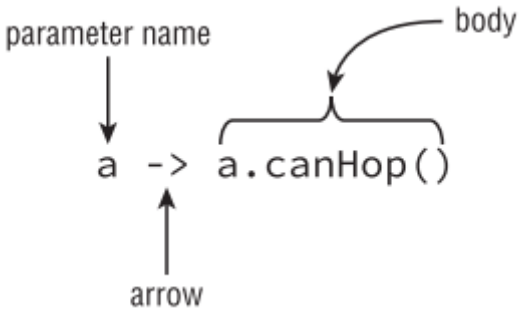
```
boolean test(Animal a);
```

Como el método de esa interfaz toma un Animal, eso significa que el parámetro lambda tiene que ser un Animal. Y dado que el método de esa interfaz arroja un valor booleano, sabemos que lambda devuelve un valor booleano.



Tenemos:

1. Especifique un solo parámetro con el nombre `a` y estableciendo que el tipo es `Animal`.
2. El operador de flecha para separar el parámetro y el cuerpo.
3. Un cuerpo que tiene una o más líneas de código, incluyendo un punto y coma y una declaración de retorno



Tenemos:

1. Especifique un solo parámetro con el nombre `a`.
2. El operador de flecha para separar el parámetro y el cuerpo.
3. Un cuerpo que llama a un solo método y devuelve el resultado de ese método.

Este atajo especial no funciona cuando tenemos dos o más declaraciones. Veamos algunos ejemplos de lambdas válidos:

```
3: print(() -> true);                                    // 0 parameters
4: print(a -> a.startsWith("test"));                     // 1 parameter
5: print((String a) -> a.startsWith("test"));            // 1 parameter
6: print((a, b) -> a.startsWith("test"));                // 2 parameters
7: print((String a, String b) -> a.startsWith("test"));  // 2 parameters
```

Ahora asegurémonos de que pueda identificar la sintaxis inválida. ¿Ves lo que está mal con cada uno de estos?

```
print(a, b -> a.startsWith("test"));                 // DOES NOT COMPILE
print(a -> { a.startsWith("test"); });               // DOES NOT COMPILE
print(a -> { return a.startsWith("test") });         // DOES NOT COMPILE
```

Es posible que haya notado que todas nuestras lambdas devuelven un booleano. Esto se debe a que el alcance del examen OCA limita lo que necesita aprender.

¿A qué variables puede acceder mi Lambda?
Lambdas tienen permitido acceder a las variables. Aquí hay un ejemplo:

```
boolean wantWhetherCanHop = true;
print(animals, a -> a.canHop() == wantWhetherCanHop);
```

El truco es que no pueden acceder a todas las variables. Instancia y variables estáticas están bien. Los parámetros del método y las variables locales están bien si no se les asignan nuevos valores.

Como Java no nos permite re declarar una variable local, el siguiente es un problema:

```
(a, b) -> { int a = 0; return 5;}      // DOES NOT COMPILE
```

Intentamos re declarar `a`, lo cual no está permitido. Por el contrario, la siguiente línea está bien porque usa un nombre de variable diferente:

```
(a, b) -> { int c = 0; return 5;}
```

8.3. Predicates

Lambdas trabaja con interfaces que tienen solo un método. Estas se llaman interfaces funcionales, interfaces que se pueden usar con programación funcional. (En realidad es más complicado que esto).

Puedes imaginar que tendríamos que crear muchas interfaces como esta para usar lambdas. Java reconoce que este es un problema común y nos proporciona una interfaz de este tipo:

```
public interface Predicate<T> {
  boolean test(T t);
}
```

Esto significa que ya no necesitamos nuestra propia interfaz y podemos poner todo lo relacionado con nuestra búsqueda en una sola clase:

```
1: import java.util.*;
2: import java.util.function.*;
3: public class PredicateSearch {
4:   public static void main(String[] args) {
5:     List<Animal> animals = new ArrayList<Animal>();
6:     animals.add(new Animal("fish", false, true));
7:
8:     print(animals, a -> a.canHop());
9:   }
10:   private static void print(List<Animal> animals, Predicate<Animal> checker)
{
11:    for (Animal animal : animals) {
12:      if (checker.test(animal))
13:         System.out.print(animal + " ");
14:     }
15:     System.out.println();
16:  }
17: }
```

Java 8 incluso integró la interfaz `Predicate` en algunas clases existentes. Solo hay uno que necesita saber para el examen. `ArrayList` declara un método `removeIf()` que toma un predicado. Decidimos que queremos eliminar todos los nombres de conejitos que no comienzan con la letra h porque nuestro pequeño primo realmente quiere que elijamos un nombre H:

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7: System.out.println(bunnies);      // [long ear, floppy, hoppy]
8: bunnies.removeIf(s -> s.charAt(0) != 'h');
9: System.out.println(bunnies);      // [hoppy]
```

La línea 8 se ocupa de todo para nosotros. Define un predicado que toma una cadena y devuelve un booleano. El método `removeIf()` hace el resto.

Para el examen OCA, solo necesita saber cómo implementar expresiones lambda que usan la interfaz `Predicate`.

9. Review Questions

9.1.

Which of the following compile? (Choose all that apply)

A. `final static void method4() { }`
B. `public final int void method() { }`
C. `private void int method() { }`
D. `static final void method3() { }`
E. `void final method() {}`
F. `void public method() { }`

9.2.

Given the following `my.school.ClassRoom` and `my.city.School` class definitions, which line numbers in `main()` generate a compiler error? (Choose all that apply)

```
1: package my.school;
2: public class Classroom {
3:   private int roomNumber;
4:   protected String teacherName;
5:   static int globalKey = 54321;
6:   public int floor = 3;
7:   Classroom(int r, String t) {
8:     roomNumber = r;
9:     teacherName = t; } }
```

```
1: package my.city;
2: import my.school.*;
3: public class School {
4:   public static void main(String[] args) {
5:     System.out.println(Classroom.globalKey);
6:     Classroom room = new Classroom(101, ""Mrs. Anderson");
7:     System.out.println(room.roomNumber);
8:     System.out.println(room.floor);
9:     System.out.println(room.teacherName); } }
```

A. None, the code compiles fine.
B. Line 5
C. Line 6
D. Line 7
E. Line 8
F. Line 9

9.3.

Which are true of the following code? (Choose all that apply)

```
1:  public class Rope {
2:    public static void swing() {
3:      System.out.print("swing ");
4:    }
5:    public void climb() {
6:      System.out.println("climb ");
7:    }
8:    public static void play() {
9:      swing();
10:     climb();
11:   }
12:   public static void main(String[] args) {
```

```
13:     Rope rope = new Rope();
14:     rope.play();
15:     Rope rope2 = null;
16:     rope2.play();
17:   }
18: }
```

A. The code compiles as is.
B. There is exactly one compiler error in the code.
C. There are exactly two compiler errors in the code.
D. If the lines with compiler errors are removed, the output is climb `climb`.
E. If the lines with compiler errors are removed, the output is swing `swing`.
F. If the lines with compile errors are removed, the code throws a `NullPointerException`.

9.4.

What is the result of the following statements?

```
1:  public class Test {
2:    public void print(byte x) {
3:      System.out.print("byte");
4:    }
5:    public void print(int x) {
6:      System.out.print("int");
7:    }
8:    public void print(float x) {
9:      System.out.print("float");
10:   }
11:   public void print(Object x) {
12:     System.out.print("Object");
13:   }
14:   public static void main(String[] args) {
15:     Test t = new Test();
16:     short s = 123;
17:     t.print(s);
18:     t.print(true);
19:     t.print(6.789);
20:   }
21: }
```

A. `bytefloatObject`
B. `intfloatObject`
C. `byteObjectfloat`
D. `intObjectfloat`
E. `intObjectObject`
F. `byteObjectObject`

9.5.

What is the result of the following?

```
1:  public class Order {
2:    static String result = "";
3:    { result += "c"; }
4:    static
5:    { result += "u"; }
6:    { result += "r"; }
7: }
1: public class OrderDriver {
2:   public static void main(String[] args) {
```

```
3:      System.out.print(Order.result + " ");
4:      System.out.print(Order.result + " ");
5:      new Order();
6:      new Order();
7:      System.out.print(Order.result + " ");
8:    }
9: }
```

A. `curur`
B. `ucrcr`
C. `u ucrcr`
D. `u u curcur`
E. `u u ucrcr`
F. `ur ur urc`
G. The code does not compile.

9.6.

Which of the following are true about the following code? (Choose all that apply)

```
public class Create {
  Create() {
    System.out.print("1 ");
  }
  Create(int num) {
    System.out.print("2 ");
  }
  Create(Integer num) {
    System.out.print("3 ");
  }
  Create(Object num) {
    System.out.print("4 ");
  }
  Create(int... nums) {
    System.out.print("5 ");
  }
  public static void main(String[] args) {
    new Create(100);
    new Create(1000L);
  }
}
```

A. The code prints out 2 4.
B. The code prints out 3 4.
C. The code prints out 4 2.
D. The code prints out 4 4.
E. The code prints 3 4 if you remove the constructor `Create(int num)`.
F. The code prints 4 4 if you remove the constructor `Create(int num)`.
G. The code prints 5 4 if you remove the constructor `Create(int num)`.

9.7.

What is the result of the following class?

```
1: import java.util.function.*;
2:
3: public class Panda {
4:   int age;
5:   public static void main(String[] args) {
6:     Panda p1 = new Panda();
```

```
7:      p1.age = 1;
8:      check(p1, p -> p.age < 5);
9:    }
10:   private static void check(Panda panda, Predicate<Panda> pred) {
11:     String result = pred.test(panda) ? "match" : "not match";
12:     System.out.print(result);
13: } }
```

A. `match`
B. `not match`
C. Compiler error on line 8.
D. Compiler error on line 10.
E. Compiler error on line 11.
F. A runtime exception is thrown.