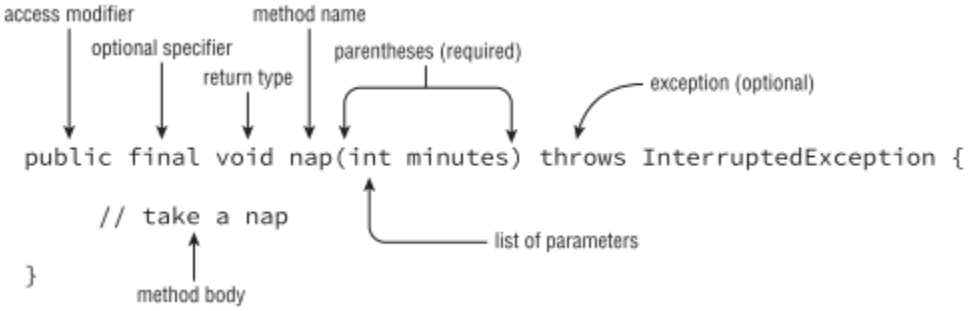
Methods And Encapsulation

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

* El cuerpo del método es omitido en métodos abstractos

1.1. 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() {}
```

1.2. 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.

1.3. 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
}
```

1.4. 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
```

1.5. 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) { }
```

1.6. 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 { }
```

1.7. 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: nums.length
```

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:

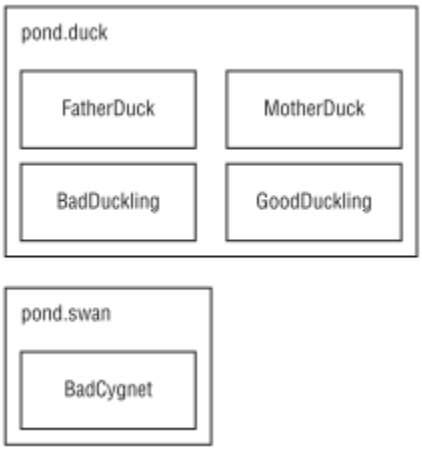
```
16: public static void run(int... nums) {
17:   System.out.println(nums[1]);
18: }
19: public static void main(String[] args) {
20:   run(11, 22);     // 22
21: }
```

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:

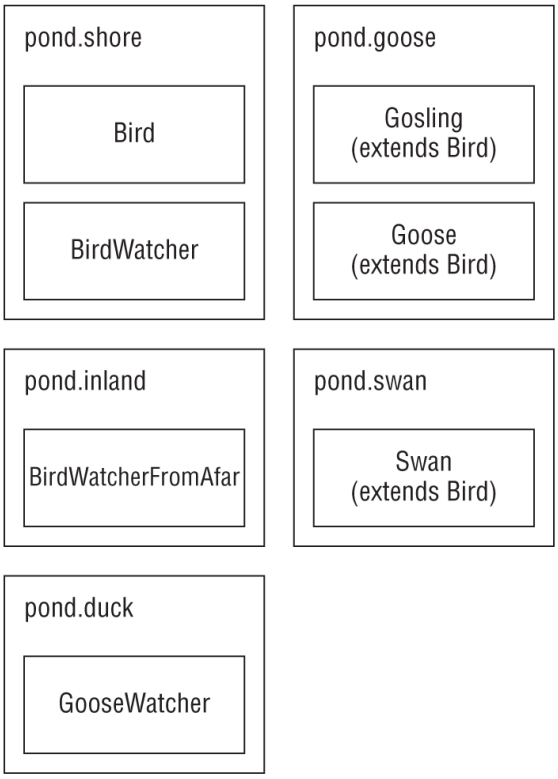
```
package pond.duck;
public class GoodDuckling {
  public void makeNoise() {
    MotherDuck duck = new MotherDuck();
    duck.quack();                          // default access
    System.out.println(duck.noise);          // default access
  } }
```

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 `protected` 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

4. Applying the static Keyword

4.1. 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.

Por ejemplo:

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

4.2. ACCESING 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
```

4.3. 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: }
```

Tipo	Llamando	¿Legal?	¿Cómo?
Método estático	Otro método estático o variable	Yes	Usando el nombre de Clase
Método estático	Un método de instancia o variable	No	
Método de Instancia	Un método estático o variable	Yes	Usando el nombre de clase o una variable de referencia
Método de Instancia	Otro método de instancia o variable	Yes	Usando una variable de referencia

¿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.

4.4. 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.

4.5. 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: }
```

4.6. 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
```

5. 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: }
```

6. 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
```

6.1. VARARGS

¿Qué método piensas que es llamado si pasamos un 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.

6.2. AUTOBOXING

¿Qué pasa si tenemos versiones con primitivos y enteros?

```
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*.

6.3. 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`.

6.4. 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.

6.5. GENERICS

Estas no son sobrecargas válidas:

```
public void walk(List<String> strings) {}
public void walk(List<Integer> integers) {}    // DOES NOT COMPILE
```

Java tiene un concepto llamado borrado de tipos (*type erasure*) donde los genéricos se usan solo en el momento de la compilación. Eso significa que el código compilado se ve así:

```
public void walk(List strings) {}
public void walk(List integers) {}    // DOES NOT COMPILE
```

Claramente, no podemos tener dos métodos con la misma firma de método, por lo que esto no se compila.

6.6. ARRAYS

A diferencia del ejemplo anterior, este código está bien:

```
public static void walk(int[] ints) {}
public static void walk(Integer[] integers) {}
```

Las matrices han existido desde el comienzo de Java. Especifican sus tipos reales y no participan en el borrado de tipos.

6.7. PUTTING IT ALL TOGETHER

El orden oficial:

Regla	**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
  } }
```

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
```

8. Review Questions

8.1.

Given the following method, which of the method calls return 2? (Choose all that apply.)

```
public int howMany(boolean b, boolean... b2) {
return b2.length;
```

}

```
A. howMany();
B. howMany(true);
C. howMany(true, true);
D. howMany(true, true, true);
E. howMany(true, {true, true});
F. howMany(true, new boolean[2]);
```

8.2.

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 static String teacherName;
5:    static int globalKey = 54321;
6:    public static 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(Classroom.floor);
9:       System.out.println(Classroom.teacherName); } }
```

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

8.3.

How many lines in the following code have compiler errors?

```
1:  public class RopeSwing {
2:     private static final String leftRope;
3:     private static final String rightRope;
4:     private static final String bench;
5:     private static final String name = "name";
6:     static {
7:        leftRope = "left";
8:        rightRope = "right";
9:     }
```

```
10:    static {
11:       name = "name";
12:       rightRope = "right";
13:    }
14:    public static void main(String[] args) {
15:       bench = "bench";
16:    }
17: }
```

A. 0
B. 1
C. 2
D. 3
E. 4
F. 5

8.4.

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. byte-float-Object-
B. int-float-Object-
C. byte-Object-float-
D. int-Object-float-
E. int-Object-Object-
F. byte-Object-Object-

8.5.

Which pairs of methods are valid overloaded pairs? (Choose all that apply.)

A. `public void hiss(Set<String> s) {}`
   and

```
   public void hiss(List<String> l) {}
B. public void baa(var c) {}
   and
   public void baa(String s) {}
C. public void meow(char ch) {}
   and
   public void meow(String s) {}
D. public void moo(char ch) {}
   and
   public void moo(char ch) {}
E. public void roar(long... longs){}
   and
   public void roar(long long) {}
F. public void woof(char... chars) {}
   and
   public void woof(Character c) {}
```