Initialization and cleanup

Initialization and cleanup

- Two of these safety issues are *initialization* and *cleanup*
  - *forget to initialize a variable*
  - *forget about an element when you’re done with it*
- Constructor
  - a special method automatically called when an object is created
- Garbage collector
  - automatically releases memory resources when they’re no longer being used

Guaranteed initialization with the constructor

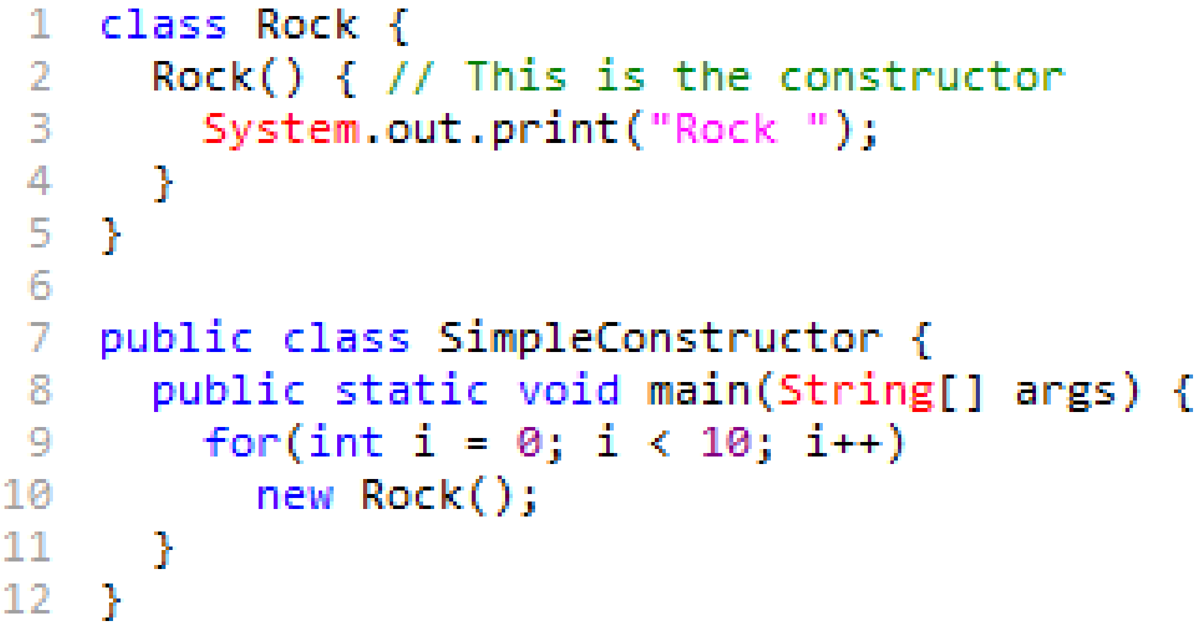
- ❑ **Java automatically calls that constructor when an object is created**

- ❑ **What to name this method?**
  - ➢ ***Clash with a name you might like to use as a member in the class***
  - ➢ ***The compiler must always know which method to call***

- ❑ **The name of the constructor is the same as the name of the class**
  - ➢ **The coding style of making the first letter of all methods lowercase does not apply to constructors**

Example

- ❑ A constructor that takes no arguments is called the ***default constructor***



```java
class Rock {
  Rock() { // This is the constructor
    System.out.print("Rock ");
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
}
```

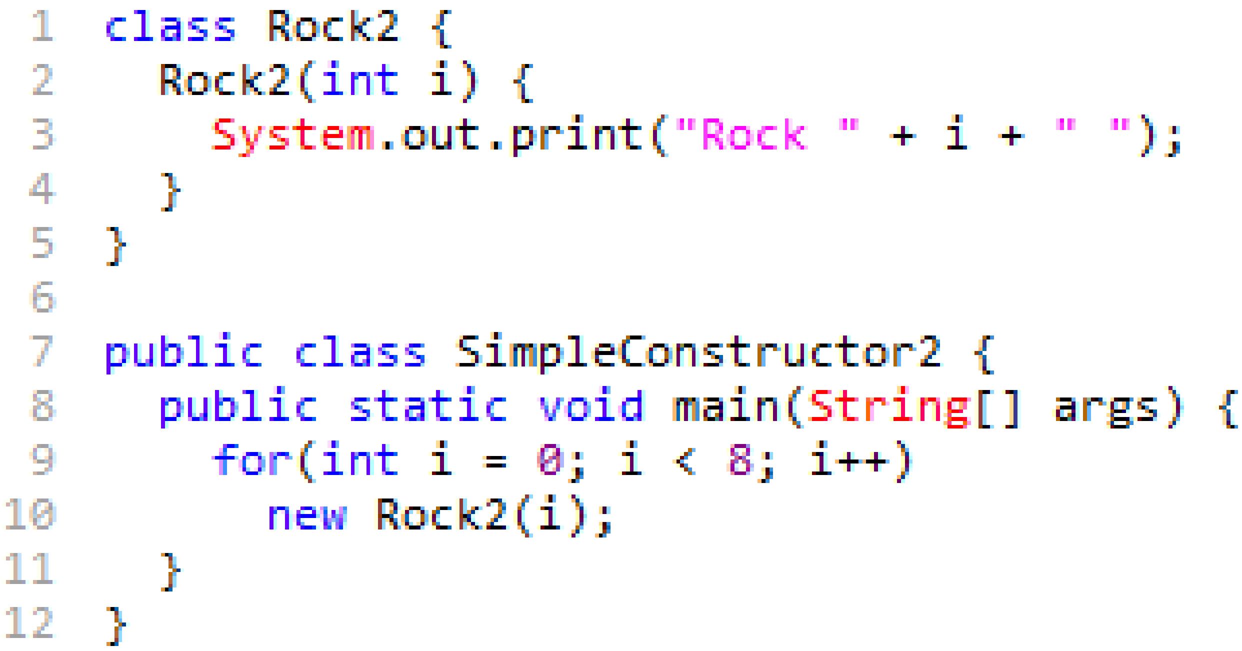
/* Output:

Rock Rock Rock Rock Rock Rock Rock Rock Rock Rock

***///:~**

Example

- ❑ **Constructor arguments provide you with a way to provide parameters for the initialization of an object**
  - ❑ ***No return value*: Different from a *void* return value**



```java
class Rock2 {
  Rock2(int i) {
    System.out.print("Rock " + i + " ");
  }
}

public class SimpleConstructor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 8; i++)
      new Rock2(i);
  }
}
```

```
/* Output:
Rock 0 Rock 1 Rock 2 Rock 3 Rock 4 Rock 5 Rock 6 Rock 7
*///:~
```

Method overloading

- ❑ You refer to all objects and methods by using names
  - ❑ Object names: give a name to a region of storage
  - ❑ Method names: a name for an action

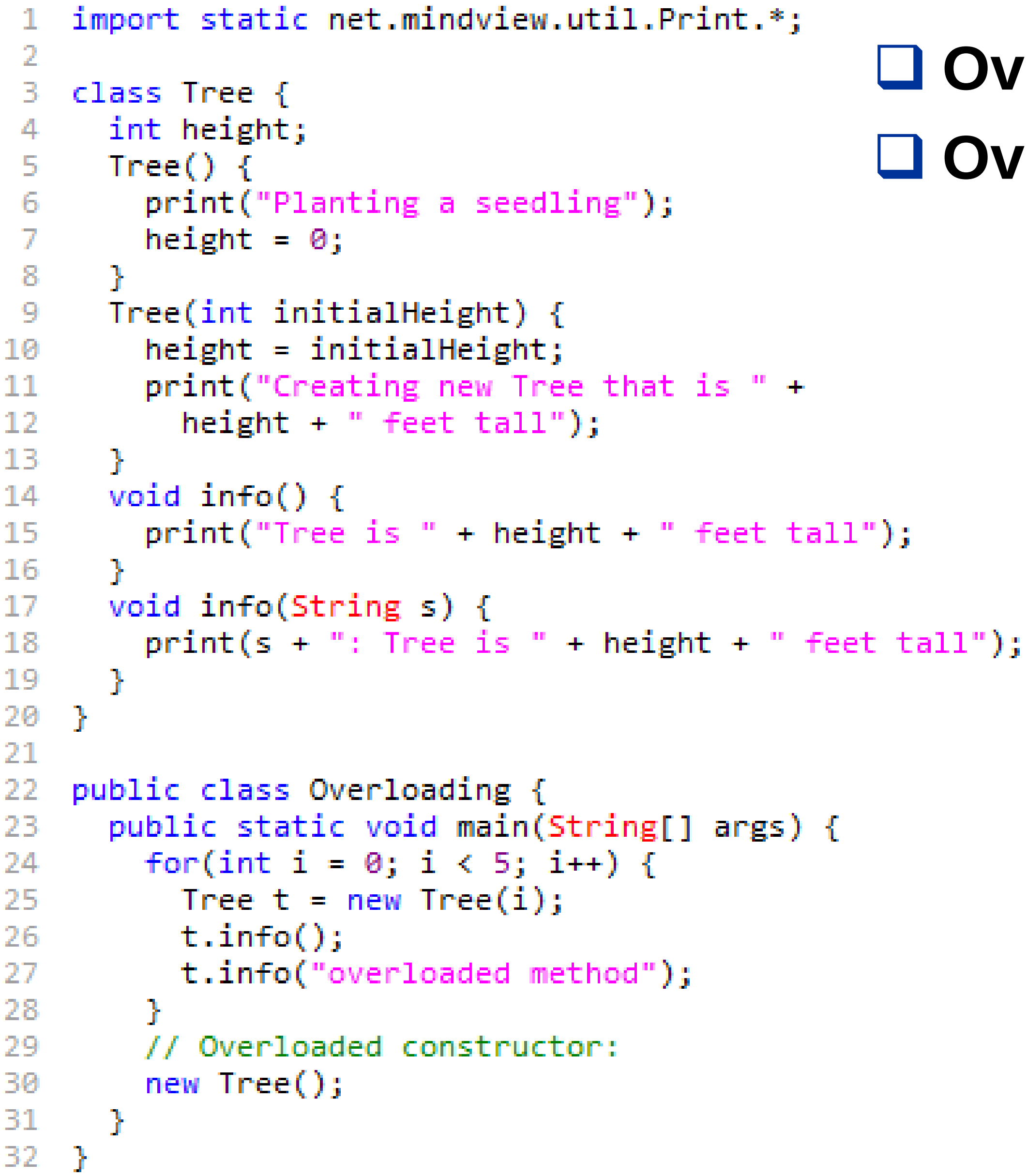
- ❑ The same word expresses a number of different meanings
  - ❑ it's *overloaded*
  - ❑ *This is useful, especially when it comes to trivial differences*
  - ❑ E.g., Wash the dog, Wash the car, Wash the shirt, etc.

- ❑ Another factor forces the overloading of method names: the constructor
  - ❑ there can be only one constructor name

Method overloading (Cont.)

- **Method overloading is essential to allow the same method name to be used with different argument types**
  - **Method overloading is a must for constructors**
  - **It can be used with any method**

Method overloading (Cont.)

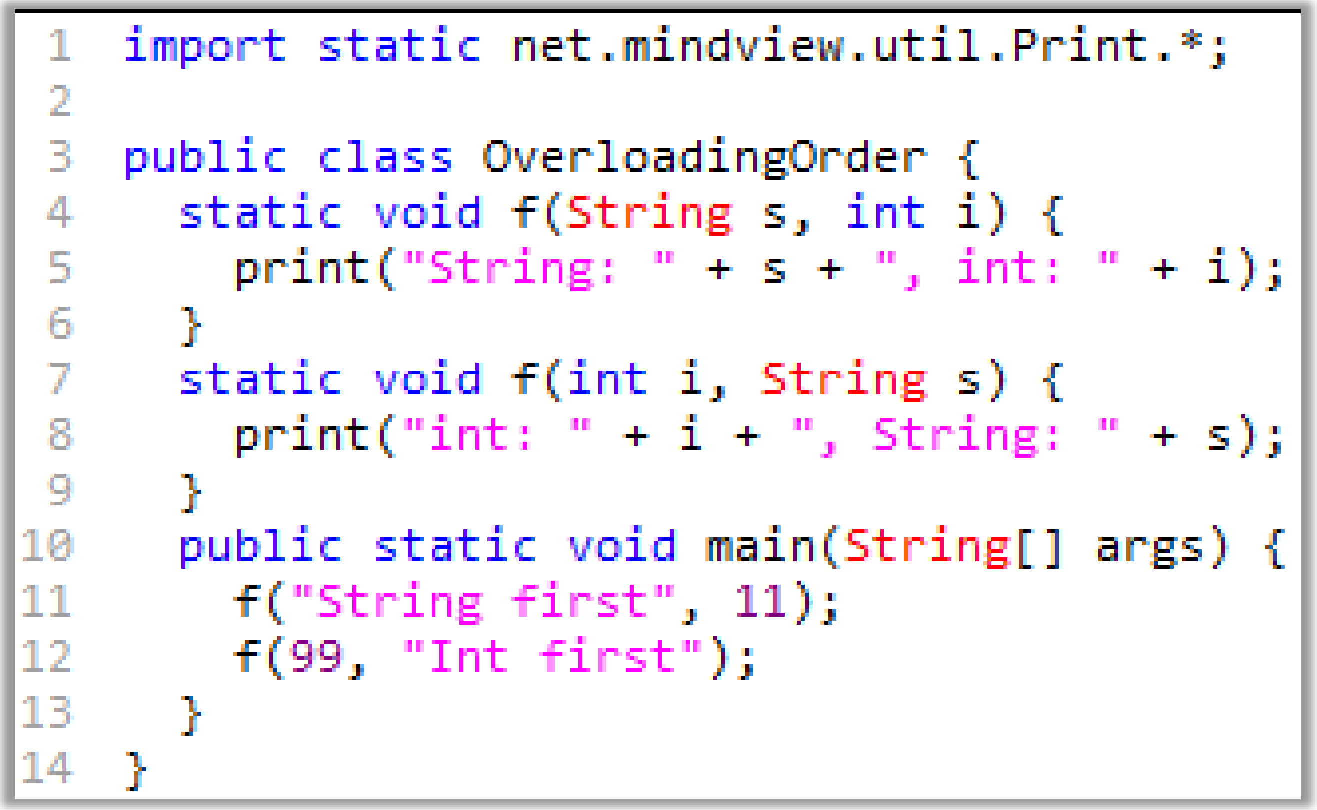


```java
import static net.mindview.util.Print.*;

class Tree {
  int height;
  Tree() {
    print("Planting a seedling");
    height = 0;
  }
  Tree(int initialHeight) {
    height = initialHeight;
    print("Creating new Tree that is " +
      height + " feet tall");
  }
  void info() {
    print("Tree is " + height + " feet tall");
  }
  void info(String s) {
    print(s + ": Tree is " + height + " feet tall");
  }
}

public class Overloading {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
  }
}
```

- ❑ Overloaded constructors
- ❑ Overloaded methods

Distinguishing overloaded methods

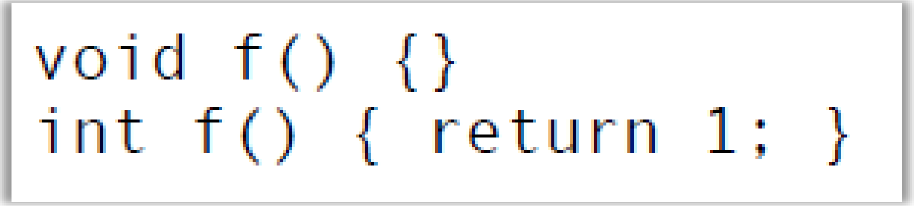
- **Simple rule: each overloaded method must take a unique list of argument types**
- **Even differences in the ordering of arguments are sufficient to distinguish two methods**



```java
import static net.mindview.util.Print.*;

public class OverloadingOrder {
  static void f(String s, int i) {
    print("String: " + s + ", int: " + i);
  }
  static void f(int i, String s) {
    print("int: " + i + ", String: " + s);
  }
  public static void main(String[] args) {
    f("String first", 11);
    f(99, "Int first");
  }
}
```

Overloading on return values

- **Why only class names and method argument lists?**
- **Why not distinguish between methods based on their return values?**
- **Example**



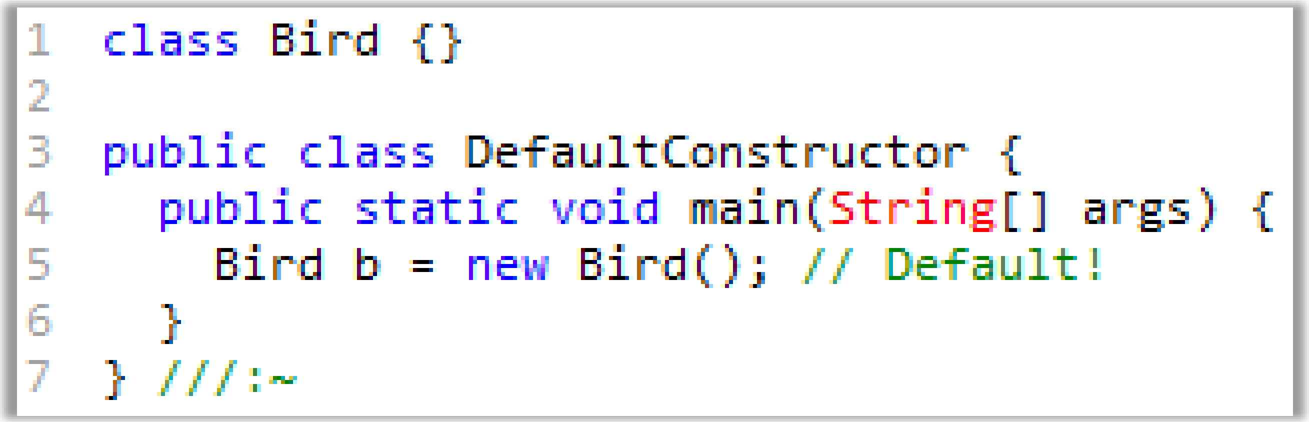
```
void f() {}
int f() { return 1; }
```

- ***Call a method for its side effect***
  - **Call a method and ignore the return value**

Default constructors

- ❑ **Default constructor (a.k.a. a “no-arg” constructor)**
  - ❑ **Create a “default object”**

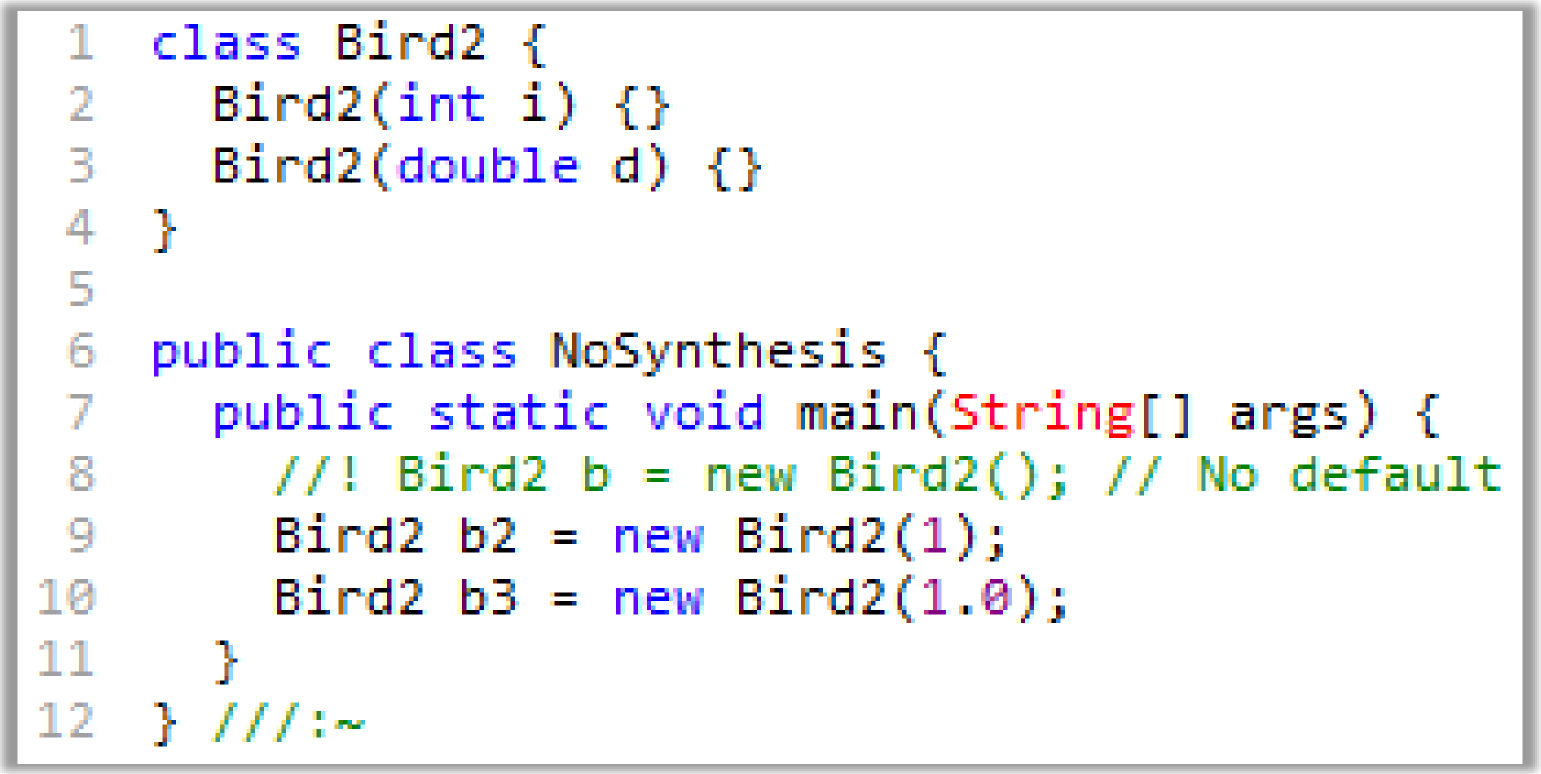
- ❑ **If you create a class that has no constructors, the compiler will automatically create a default constructor for you**



```java
class Bird {}

public class DefaultConstructor {
  public static void main(String[] args) {
    Bird b = new Bird(); // Default!
  }
} ///:~
```

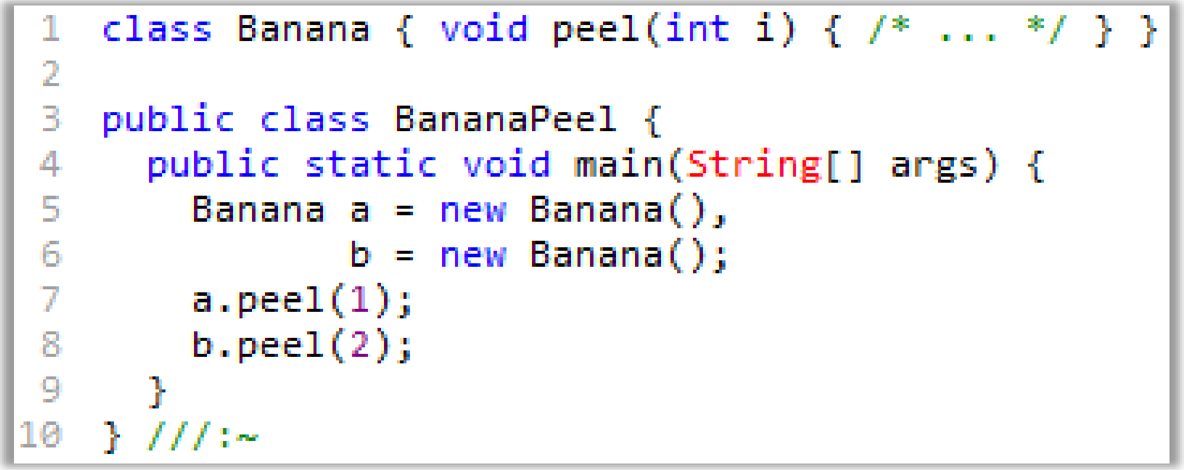
Default constructors (Cont.)

- If you define any constructors (with or without arguments), the compiler will not synthesize one for you



```java
class Bird2 {
  Bird2(int i) {}
  Bird2(double d) {}
}

public class NoSynthesis {
  public static void main(String[] args) {
    //! Bird2 b = new Bird2(); // No default
    Bird2 b2 = new Bird2(1);
    Bird2 b3 = new Bird2(1.0);
  }
} ///:~
```

How can a method know whether it's being called for different objects?



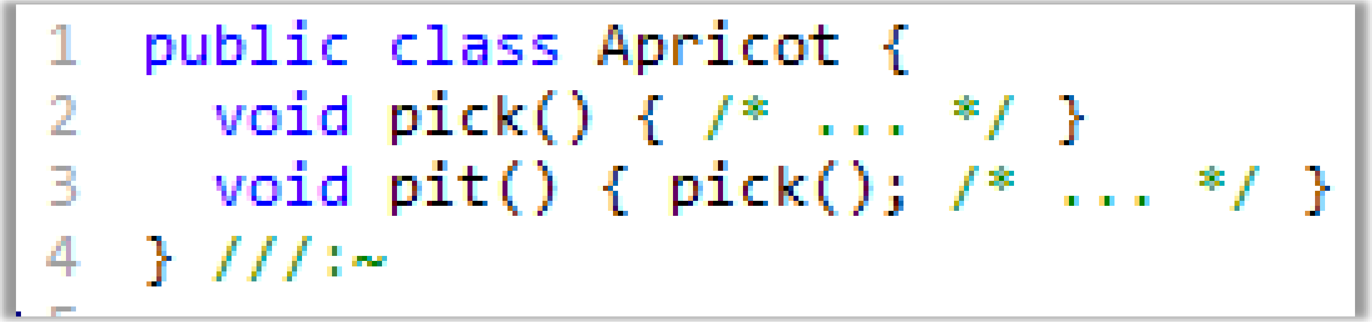
```java
class Banana { void peel(int i) { /* ... */ } }

public class BananaPeel {
  public static void main(String[] args) {
    Banana a = new Banana(),
           b = new Banana();
    a.peel(1);
    b.peel(2);
  }
} ///:~
```

- ❑ **The compiler does some undercover work for you**
  - ❑ **Write the code in a convenient object-oriented syntax**
  - ❑ **"send a message to an object"**
- ❑ **There's a secret first argument passed to the method**

```java
Banana.peel(a, 1);
Banana.peel(b, 2);
```

The *this* keyword

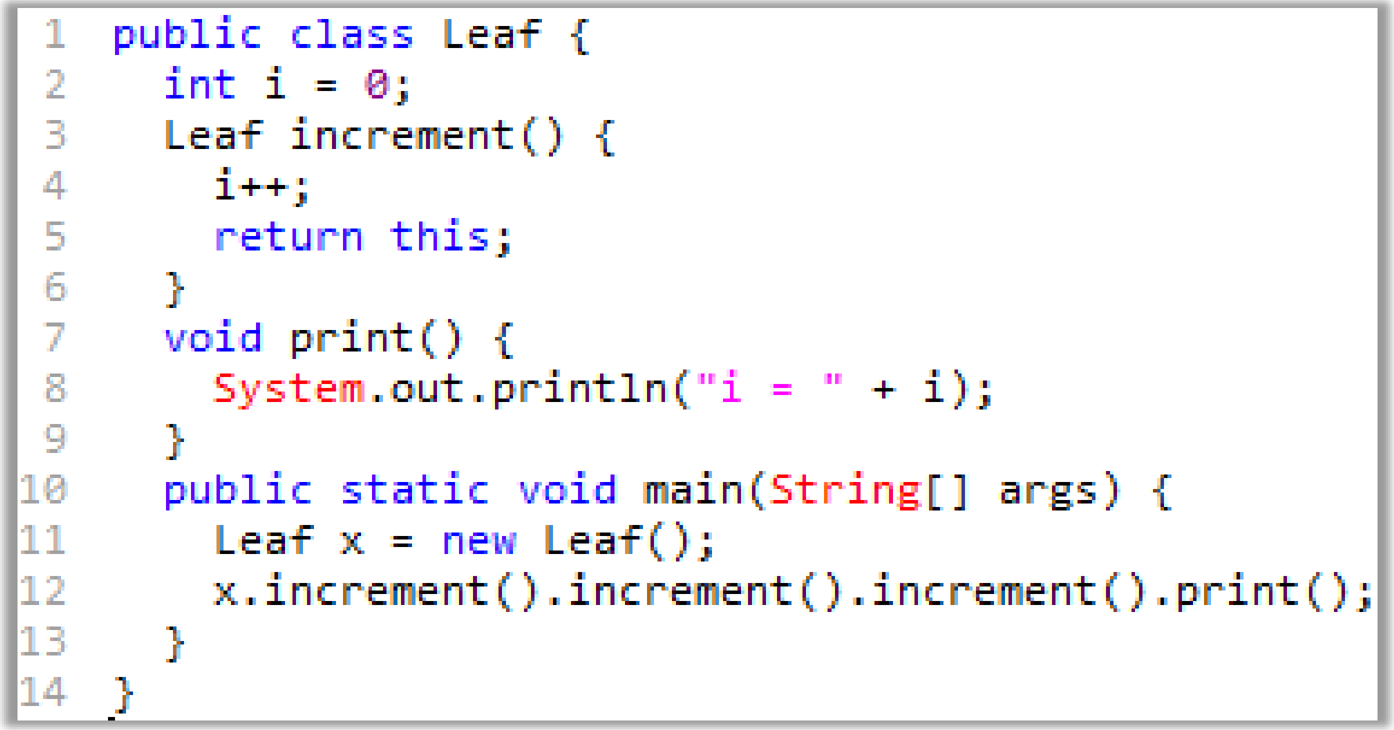
- Reference is passed *secretly* by the compiler, there’s no identifier for it
- The *this* keyword produces the reference to the object that the method has been called for
- Call a method of your class from within another method of your class
  - you *don’t* need to use *this*
- *this* can be used only inside a non-static method



```
public class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
} ///:~
```

The *this* keyword (Cont.)

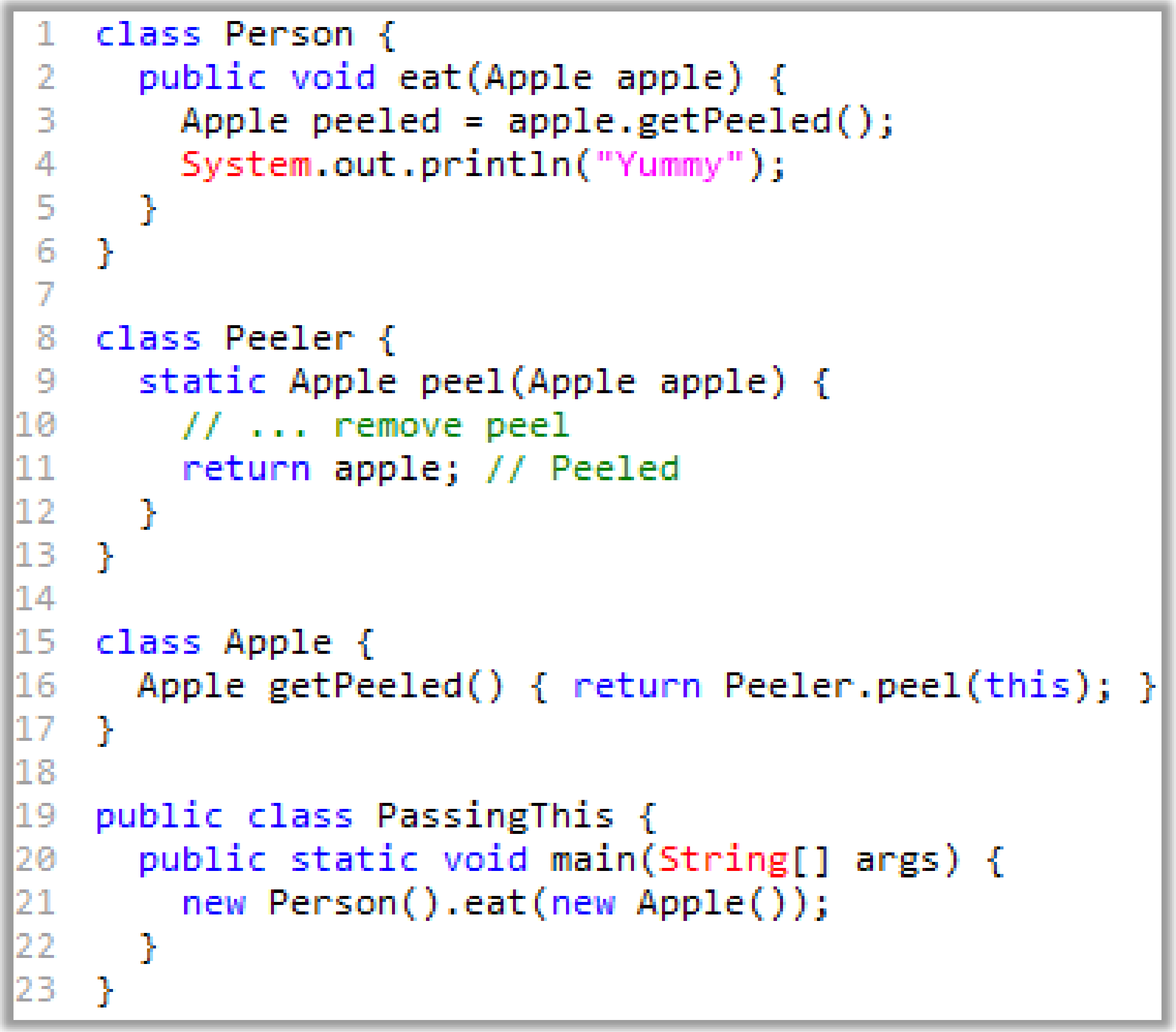
- The ***this*** keyword is used only for those special cases in which you need to ***explicitly use the reference to the current object***
- **Example**
  - **used in *return* statements when you want to return the reference to the current object**



```
public class Leaf {
  int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
}
```

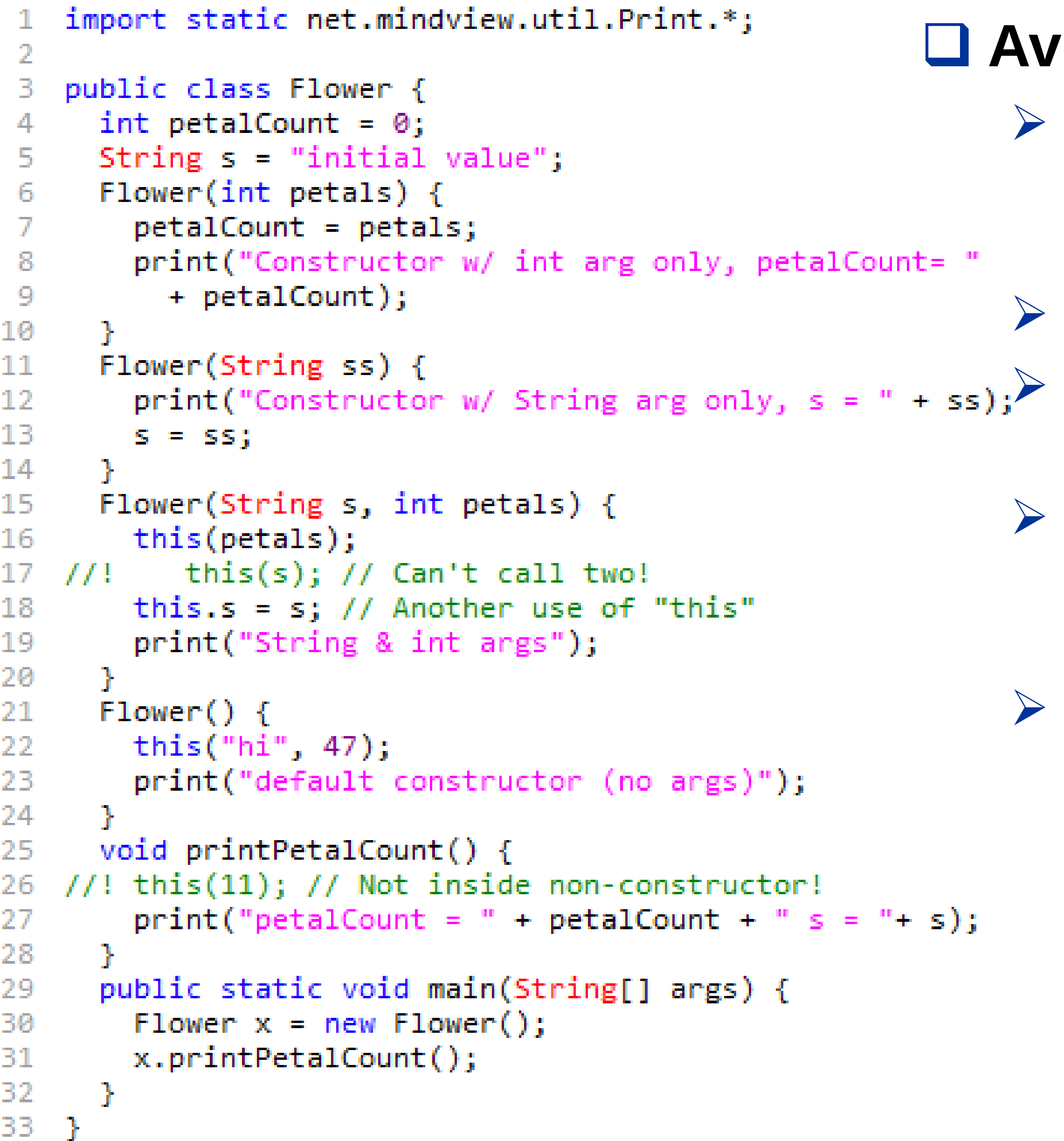
The *this* keyword (Cont.)

- **Pass the current object to another method**
  - **To pass itself to the foreign method, it must use *this***



```java
class Person {
  public void eat(Apple apple) {
    Apple peeled = apple.getPeeled();
    System.out.println("Yummy");
  }
}

class Peeler {
  static Apple peel(Apple apple) {
    // ... remove peel
    return apple; // Peeled
  }
}

class Apple {
  Apple getPeeled() { return Peeler.peel(this); }
}

public class PassingThis {
  public static void main(String[] args) {
    new Person().eat(new Apple());
  }
}
```

Calling constructors from constructors



```
import static net.mindview.util.Print.*;

public class Flower {
  int petalCount = 0;
  String s = "initial value";
  Flower(int petals) {
    petalCount = petals;
    print("Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(String ss) {
    print("Constructor w/ String arg only, s = " + ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
//!    this(s); // Can't call two!
    this.s = s; // Another use of "this"
    print("String & int args");
  }
  Flower() {
    this("hi", 47);
    print("default constructor (no args)");
  }
  void printPetalCount() {
//! this(11); // Not inside non-constructor!
    print("petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.printPetalCount();
  }
}
```

- **Avoid duplicating code**
  - **Call one constructor from another by using the *this* keyword**
  - **Cannot call two *this***
  - **The constructor call must be the first thing you do**
  - **Cannot call a constructor from inside any method other than a constructor**
  - **There is no *this* for static method**

Cleanup: finalization and garbage collection

- The garbage collector only knows how to release memory allocated with *new*, so it won't know how to release the object's "special" memory

- Java provides a method called *finalize()* that you can define for your class
  - Garbage collector is ready to release the storage used for your object, it will call *finalize( )*
  - On next garbage-collection pass it will reclaim the object's memory

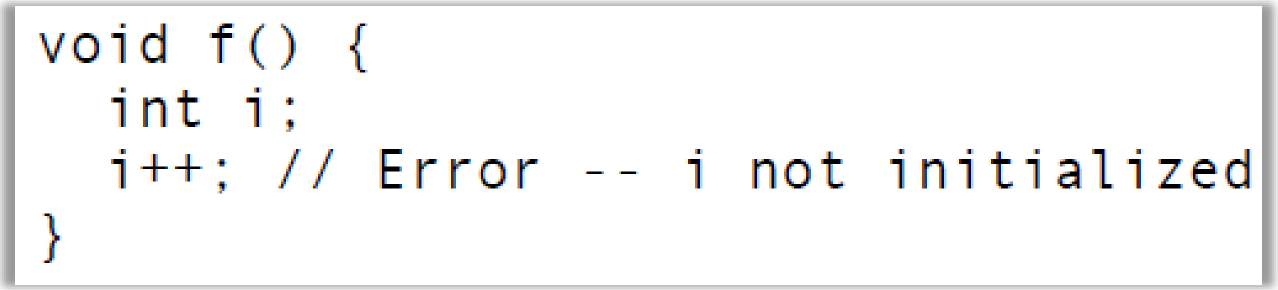
- *finalize( )* gives you the ability to perform some important cleanup at the time of garbage collection

Cleanup: finalization and garbage collection (Cont.)

- C++ programmers, might initially mistake *finalize()* for the *destructor* in C++, which is a function that is always called when an object is destroyed
  - C++: Objects always get *destroyed* (in a bug-free program)
  - Java: Objects do not always get garbage collected
- *Note:*
  - Your objects might not get garbage collected
  - Garbage collection is not destruction
  - Garbage collection is only about memory
- Remember that neither garbage collection nor finalization is guaranteed
  - If the JVM isn’t close to running out of memory, then it might not waste time recovering memory through garbage collection
  - you can’t rely on *finalize( )* being called, and you must create separate “cleanup” methods and call them explicitly

Member initialization

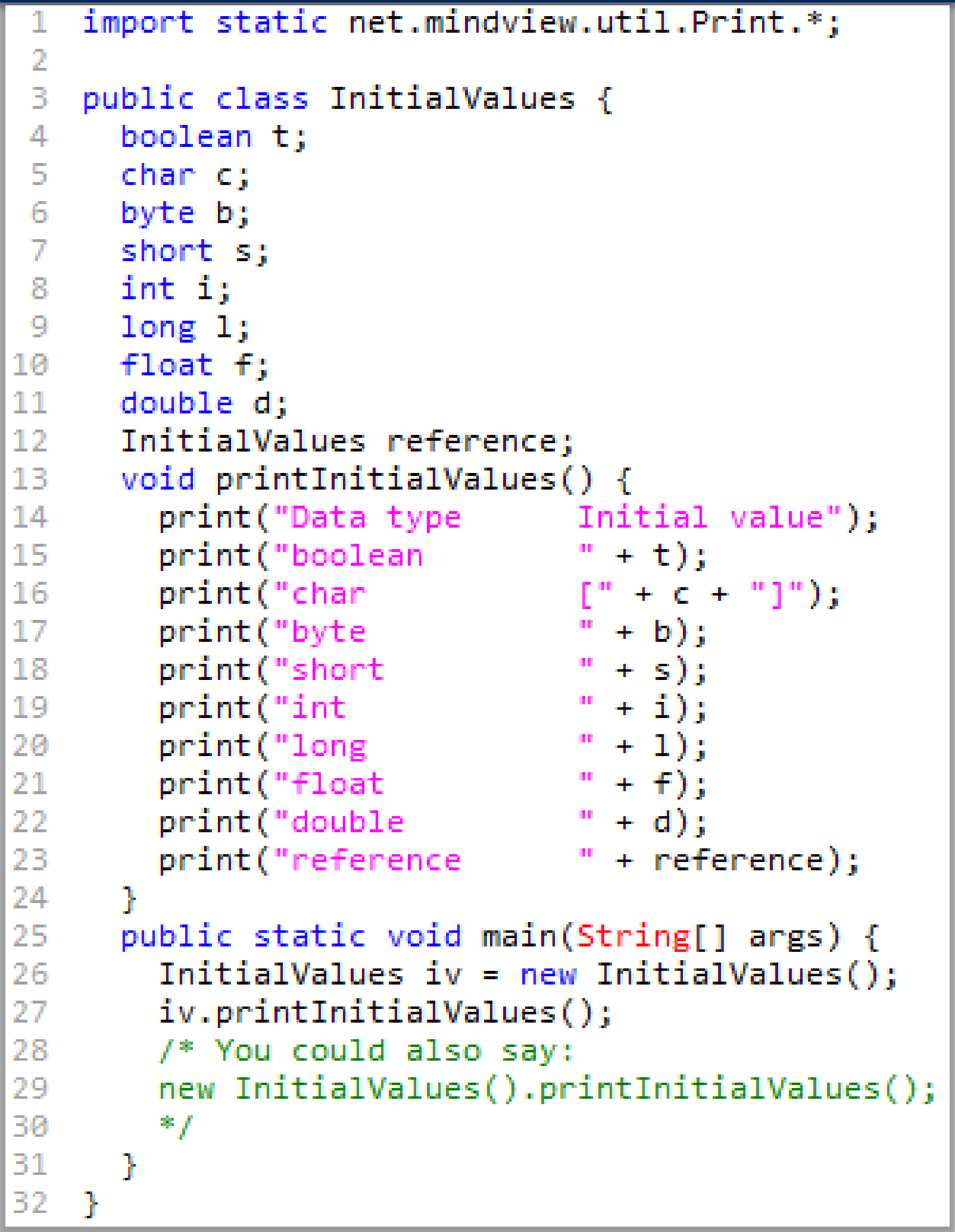
- **Java guarantee that variables are properly initialized before they are used**
  - **A method’s local variables are not initialized (*compiler-time error*)**



```
void f() {
  int i;
  i++; // Error -- i not initialized
}
```

  - **Each primitive field of a class is guaranteed to get an initial value**

Member initialization (Cont.)

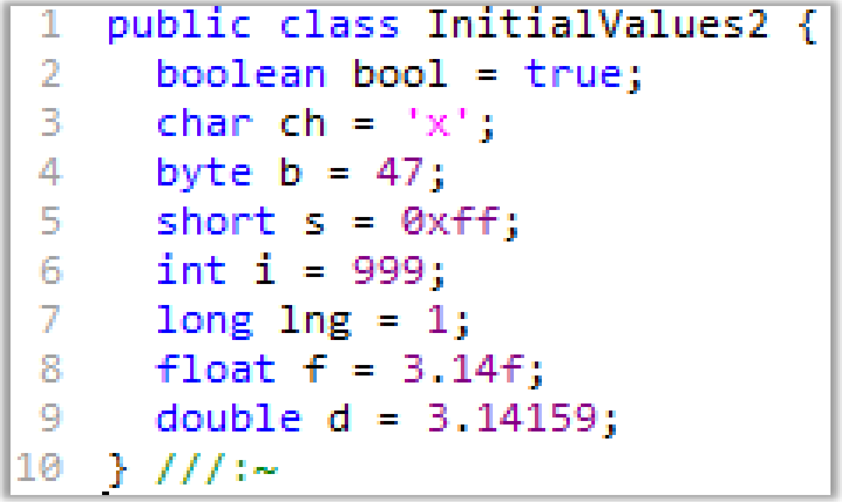


```java
import static net.mindview.util.Print.*;

public class InitialValues {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  InitialValues reference;
  void printInitialValues() {
    print("Data type      Initial value");
    print("boolean        " + t);
    print("char           [" + c + "]");
    print("byte           " + b);
    print("short          " + s);
    print("int            " + i);
    print("long           " + l);
    print("float          " + f);
    print("double         " + d);
    print("reference      " + reference);
  }
  public static void main(String[] args) {
    InitialValues iv = new InitialValues();
    iv.printInitialValues();
    /* You could also say:
    new InitialValues().printInitialValues();
    */
  }
}
```

- **Each primitive field of a class is guaranteed to get an initial value**
  - **The *char* value is a zero, which prints as a space**
  - **The reference is given a special value of *null***

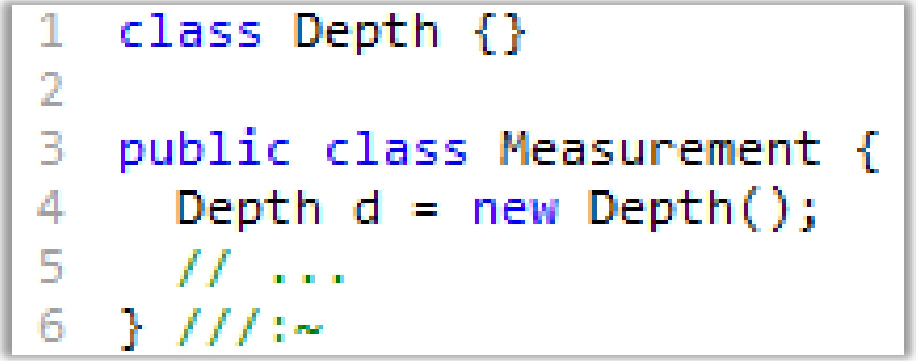
Specifying initialization

- ❑ Assign the value at the point you define the variable in the class



```
public class InitialValues2 {
  boolean bool = true;
  char ch = 'x';
  byte b = 47;
  short s = 0xff;
  int i = 999;
  long lng = 1;
  float f = 3.14f;
  double d = 3.14159;
} ///:~
```

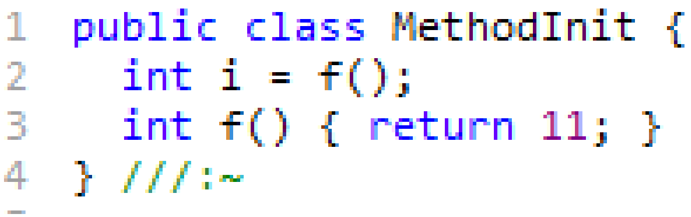
- ❑ Initialize non-primitive objects in this same way
  - ❑ Get a *runtime error* called an ***exception***, if it is not initialized and you try to use it anyway



```
class Depth {}

public class Measurement {
  Depth d = new Depth();
  // ...
} ///:~
```

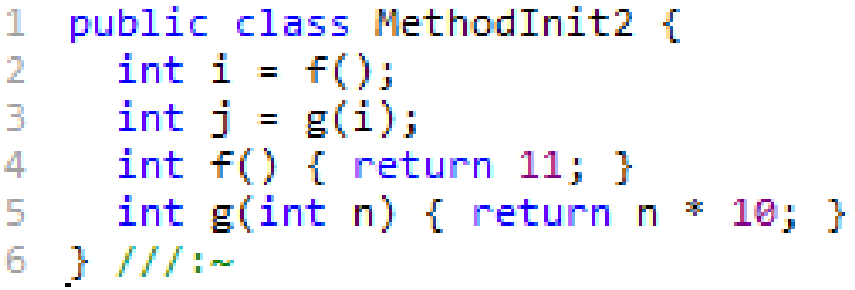
Specifying initialization (Cont.)

- ❑ Call a method to provide an initialization value

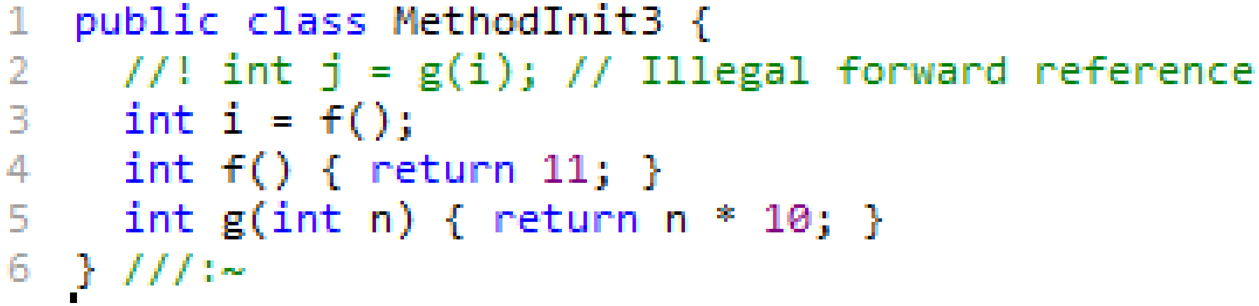


```java
public class MethodInit {
  int i = f();
  int f() { return 11; }
} ///:~
```

- ❑ This method can have arguments, but those arguments cannot be other class members that haven’t been initialized yet



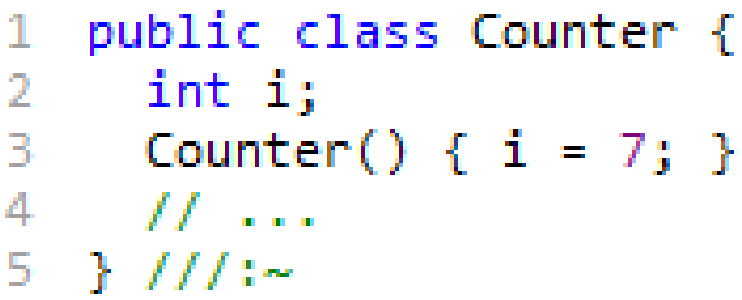
```java
public class MethodInit2 {
  int i = f();
  int j = g(i);
  int f() { return 11; }
  int g(int n) { return n * 10; }
} ///:~
```



```java
public class MethodInit3 {
  //! int j = g(i); // Illegal forward reference
  int i = f();
  int f() { return 11; }
  int g(int n) { return n * 10; }
} ///:~
```

Constructor initialization

- The constructor can be used to perform initialization
  - Call methods and perform actions at run time to determine the initial values
  - You *cannot* preclude the automatic initialization



```
public class Counter {
  int i;
  Counter() { i = 7; }
  // ...
} ///:~
```

  - The compiler doesn’t try to force you to initialize elements in the constructor at any particular place, or before they are used—initialization is already guaranteed

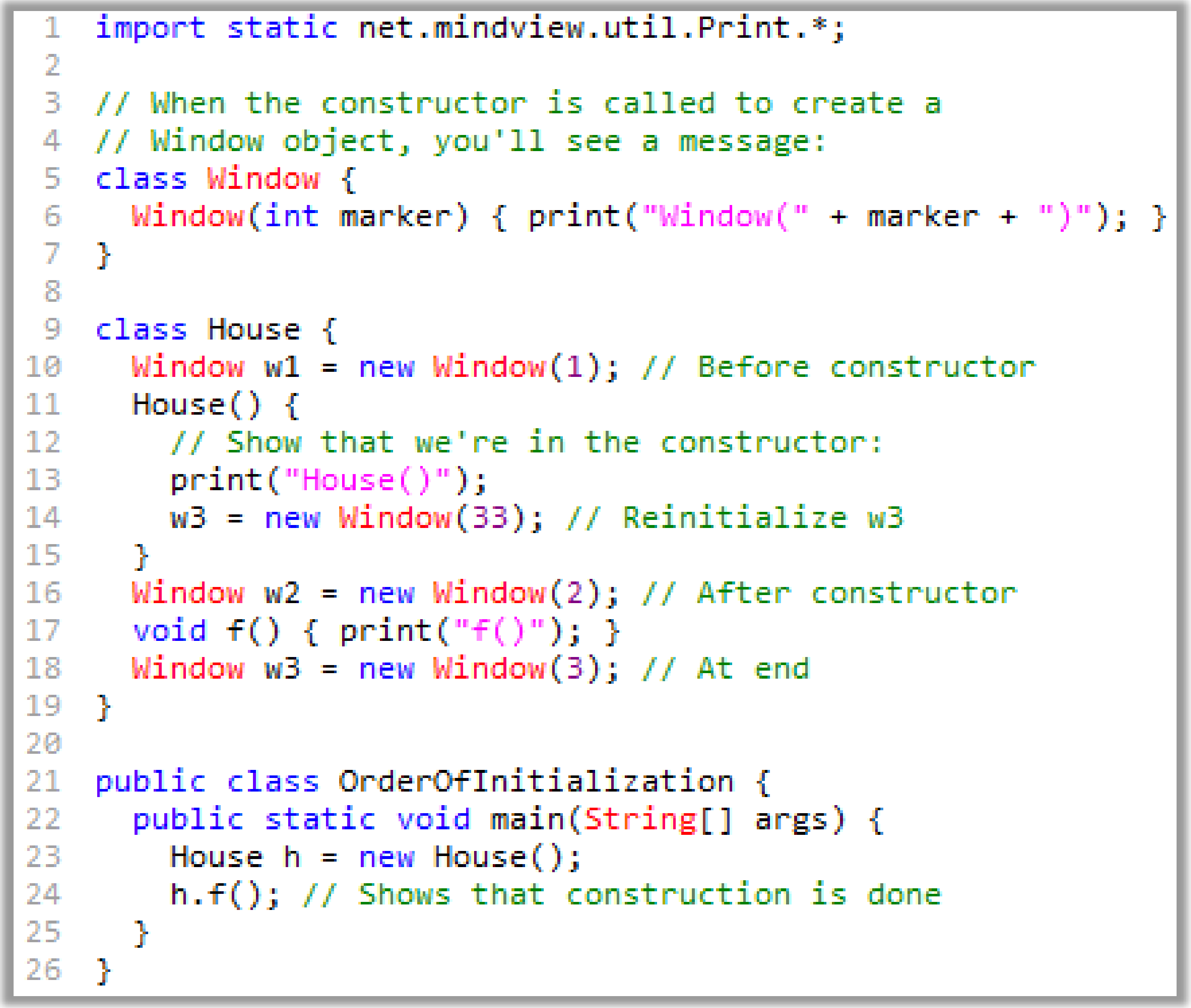
Order of initialization

- ❑The order of initialization is determined by the order that the variables are defined within the class

- ❑The variable definitions may be scattered throughout and in between method definitions

- ❑The variables are initialized before any methods can be called—even the constructor

Order of initialization



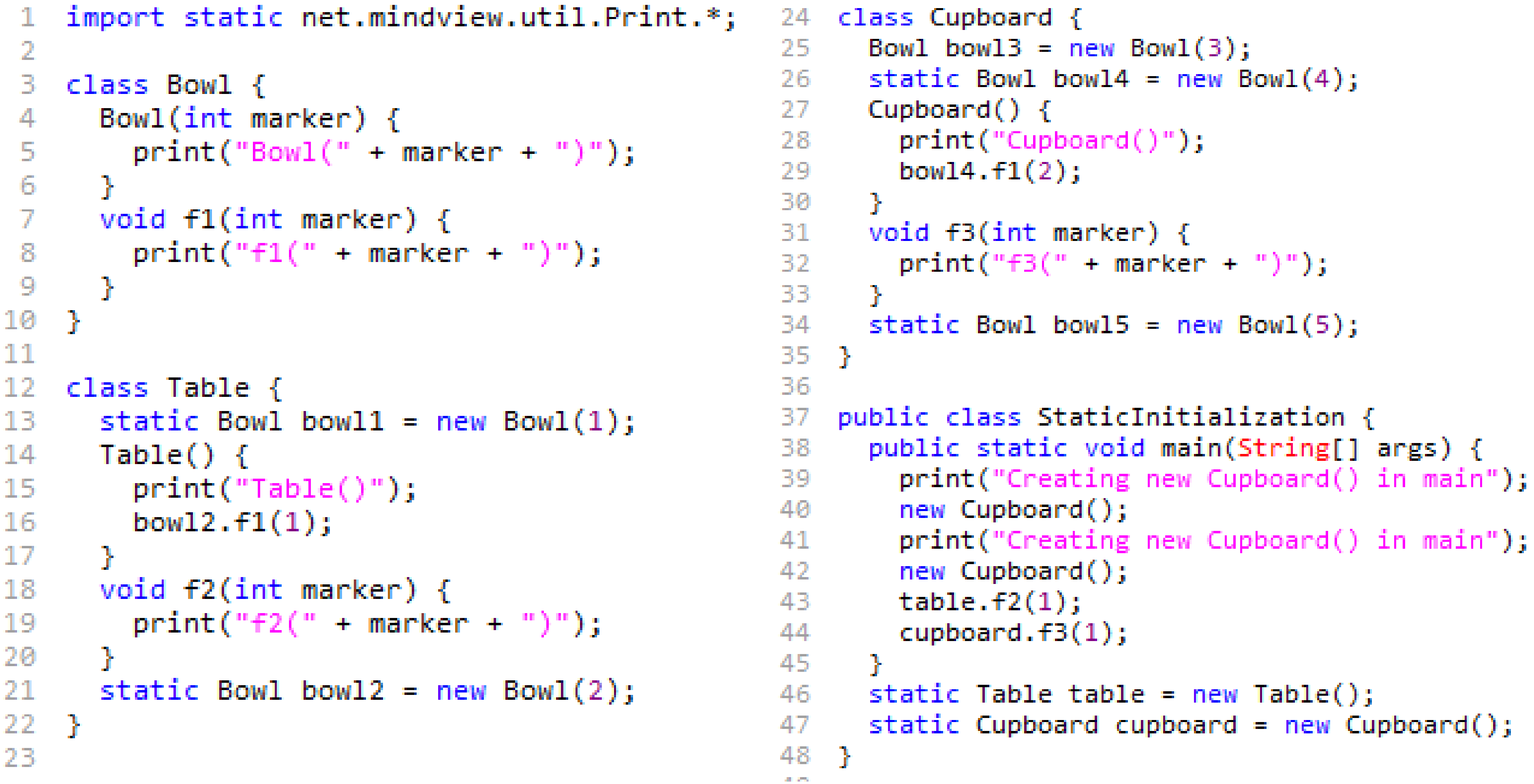
```java
import static net.mindview.util.Print.*;

// When the constructor is called to create a
// Window object, you'll see a message:
class Window {
  Window(int marker) { print("Window(" + marker + ")"); }
}

class House {
  Window w1 = new Window(1); // Before constructor
  House() {
    // Show that we're in the constructor:
    print("House()");
    w3 = new Window(33); // Reinitialize w3
  }
  Window w2 = new Window(2); // After constructor
  void f() { print("f()"); }
  Window w3 = new Window(3); // At end
}

public class OrderOfInitialization {
  public static void main(String[] args) {
    House h = new House();
    h.f(); // Shows that construction is done
  }
}
```

static data initialization

- *static* only applies to fields
  - Cannot apply to local variables
- *static* primitive: get the standard initial value for its type
- *static* reference: the default initialization value is *null*

- They are initialized only when the first object is created (or the first static access occurs)
  - After that, the static objects are not reinitialized
  - The order of initialization is *static*s first, and then the non-static objects

static data initialization

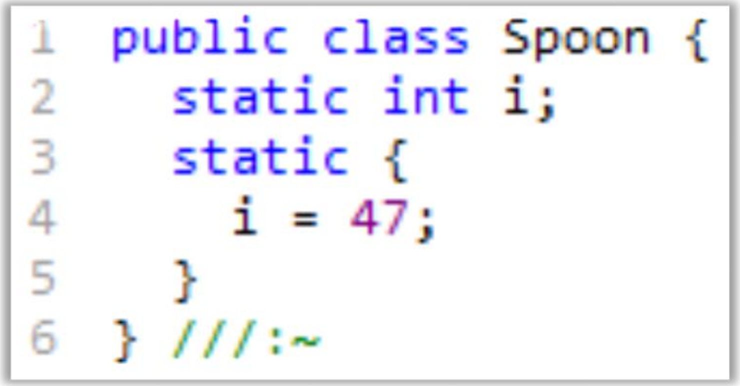


```java
import static net.mindview.util.Print.*;

class Bowl {
  Bowl(int marker) {
    print("Bowl(" + marker + ")");
  }
  void f1(int marker) {
    print("f1(" + marker + ")");
  }
}

class Table {
  static Bowl bowl1 = new Bowl(1);
  Table() {
    print("Table()");
    bowl2.f1(1);
  }
  void f2(int marker) {
    print("f2(" + marker + ")");
  }
  static Bowl bowl2 = new Bowl(2);
}

class Cupboard {
  Bowl bowl3 = new Bowl(3);
  static Bowl bowl4 = new Bowl(4);
  Cupboard() {
    print("Cupboard()");
    bowl4.f1(2);
  }
  void f3(int marker) {
    print("f3(" + marker + ")");
  }
  static Bowl bowl5 = new Bowl(5);
}

public class StaticInitialization {
  public static void main(String[] args) {
    print("Creating new Cupboard() in main");
    new Cupboard();
    print("Creating new Cupboard() in main");
    new Cupboard();
    table.f2(1);
    cupboard.f3(1);
  }
  static Table table = new Table();
  static Cupboard cupboard = new Cupboard();
}
```

Explicit *static* initialization

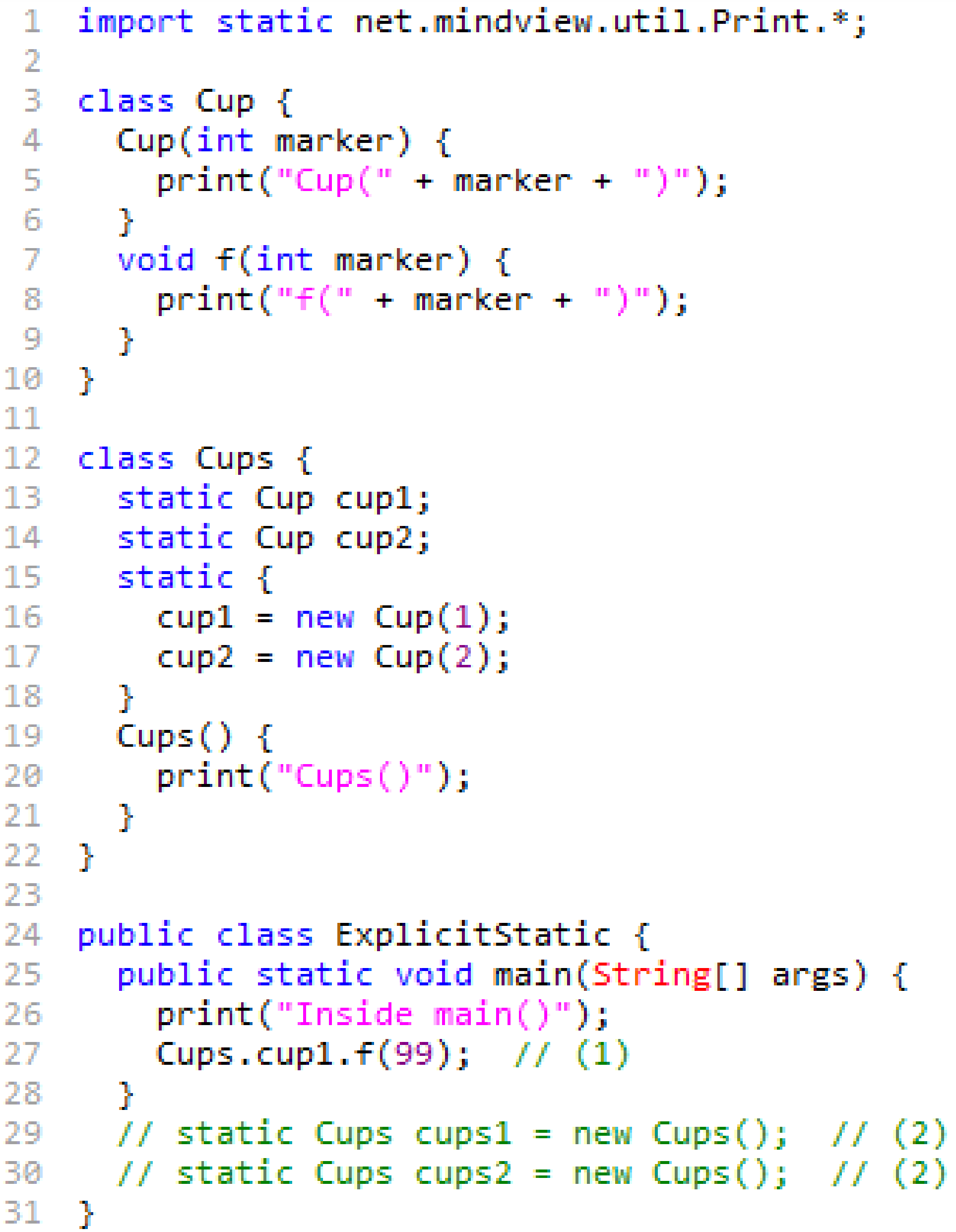
- Java allows you to group other static initializations inside a special "*static clause*" (sometimes called a *static block*) in a class

- Like other *static* initializations, it is executed only once
  - The first time you make an object of that class
  - Or the first time you access a static member of that class



```
public class Spoon {
  static int i;
  static {
    i = 47;
  }
} ///:~
```

Explicit *static* initialization

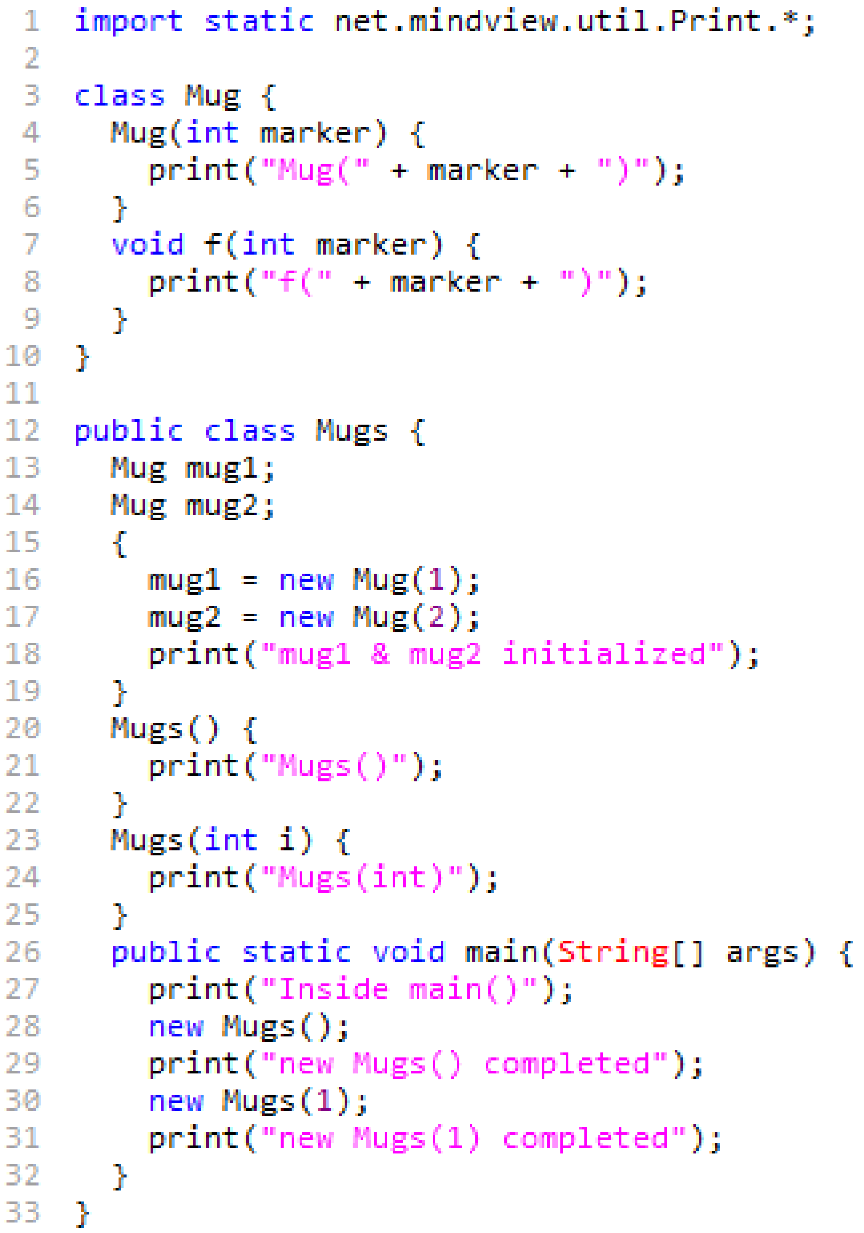


```java
import static net.mindview.util.Print.*;

class Cup {
  Cup(int marker) {
    print("Cup(" + marker + ")");
  }
  void f(int marker) {
    print("f(" + marker + ")");
  }
}

class Cups {
  static Cup cup1;
  static Cup cup2;
  static {
    cup1 = new Cup(1);
    cup2 = new Cup(2);
  }
  Cups() {
    print("Cups()");
  }
}

public class ExplicitStatic {
  public static void main(String[] args) {
    print("Inside main()");
    Cups.cup1.f(99);  // (1)
  }
  // static Cups cups1 = new Cups();  // (2)
  // static Cups cups2 = new Cups();  // (2)
}
```

Non-static instance initialization

- Java provides a similar syntax, called ***instance initialization***, for initializing non-static variables for each object

Non-static instance initialization



```java
import static net.mindview.util.Print.*;

class Mug {
  Mug(int marker) {
    print("Mug(" + marker + ")");
  }
  void f(int marker) {
    print("f(" + marker + ")");
  }
}

public class Mugs {
  Mug mug1;
  Mug mug2;
  {
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    print("mug1 & mug2 initialized");
  }
  Mugs() {
    print("Mugs()");
  }
  Mugs(int i) {
    print("Mugs(int)");
  }
  public static void main(String[] args) {
    print("Inside main()");
    new Mugs();
    print("new Mugs() completed");
    new Mugs(1);
    print("new Mugs(1) completed");
  }
}
```

- ❑ **This syntax is necessary to support the initialization of *anonymous inner classes***

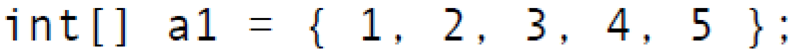
Array initialization

- An array is simply a sequence of either objects or primitives that are all the same type and are packaged together under one identifier name
  - Arrays are defined and used with the square-brackets *indexing operator [ ]*

```
int[] a1;        int a1[];
```

- The compiler doesn’t allow you to tell it how big the array is
  - All that you have at this point is a reference to an array

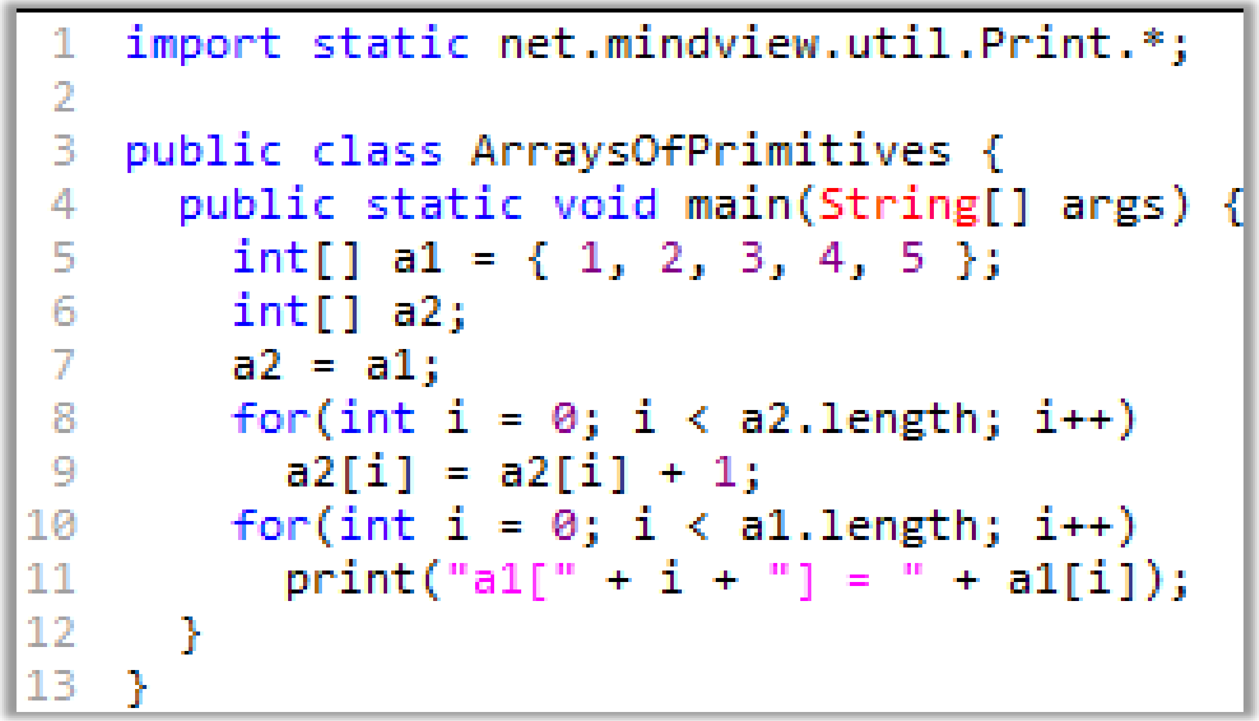
- A special initialization is a set of values surrounded by curly braces
  - The storage allocation (the equivalent of using *new*) is taken care of by the compiler in this case



```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Array initialization – first form

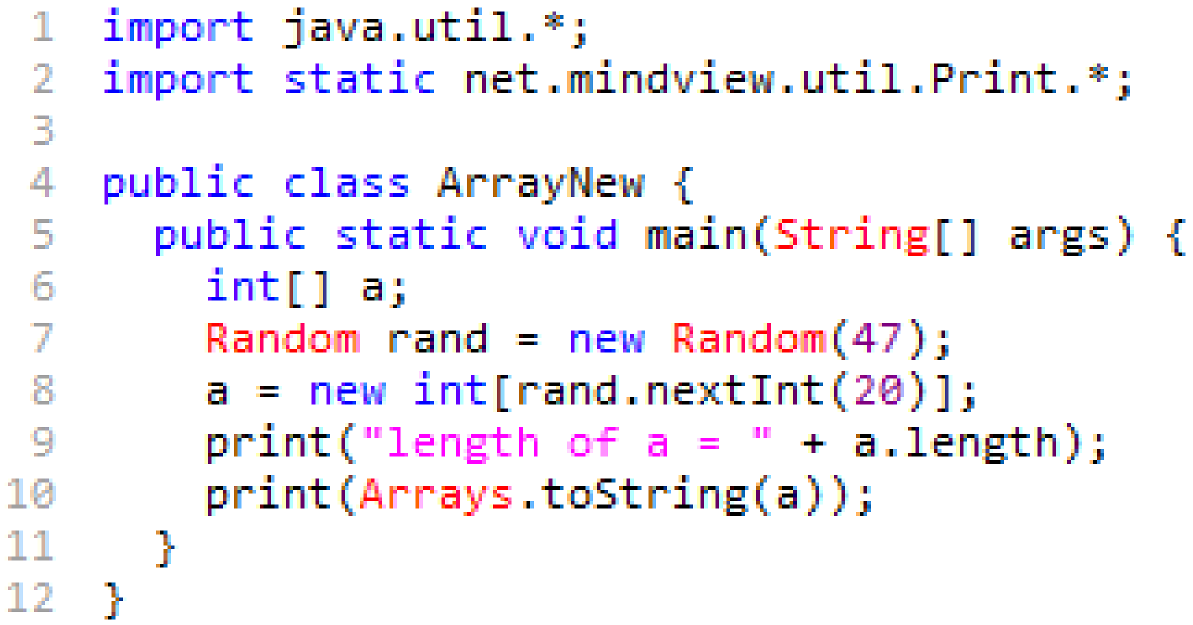
- Why would you ever define an array reference without an array?
  - It’s possible to assign one array to another in Java
  - What you’re really doing is copying a ***reference***



```java
import static net.mindview.util.Print.*;

public class ArraysOfPrimitives {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i] = a2[i] + 1;
    for(int i = 0; i < a1.length; i++)
      print("a1[" + i + "] = " + a1[i]);
  }
}
```

Array initialization – second form

- ❑ **What if you don't know how many elements you're going to need in your array while you're writing the program?**
  - ❑ **You simply use *new* to create the elements in the array**

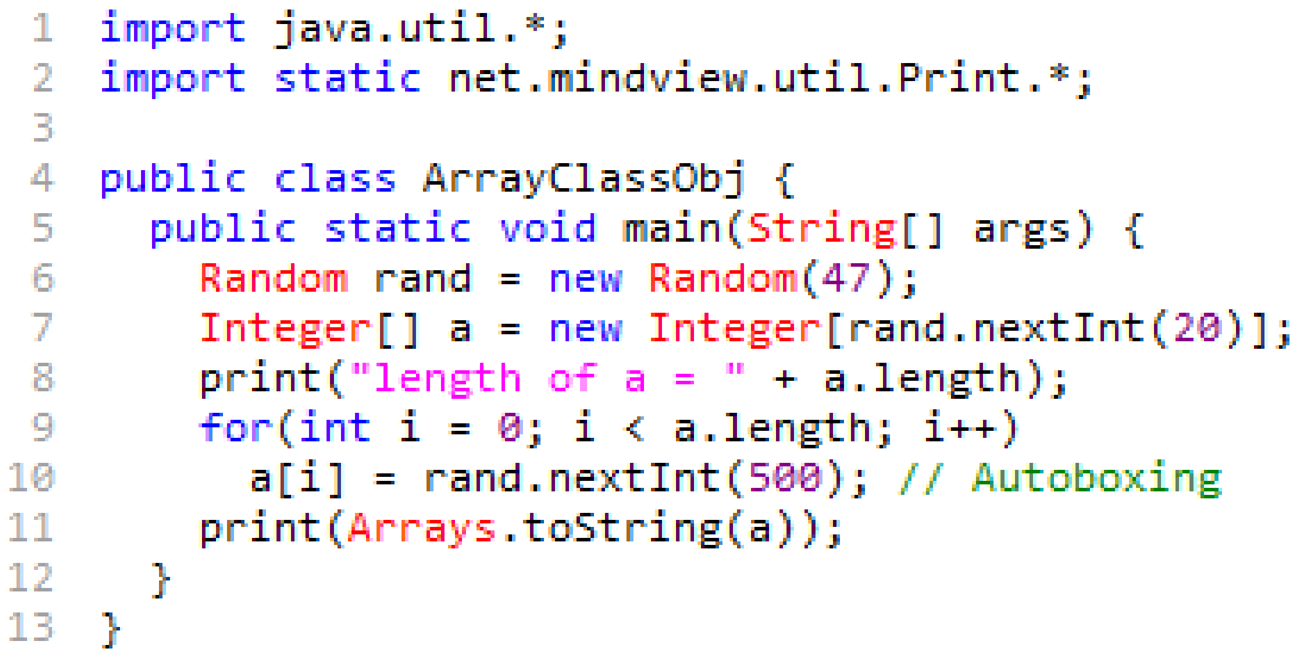


```java
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayNew {
  public static void main(String[] args) {
    int[] a;
    Random rand = new Random(47);
    a = new int[rand.nextInt(20)];
    print("length of a = " + a.length);
    print(Arrays.toString(a));
  }
}
```

```java
int[] a = new int[rand.nextInt(20)];
```

Array initialization – second form

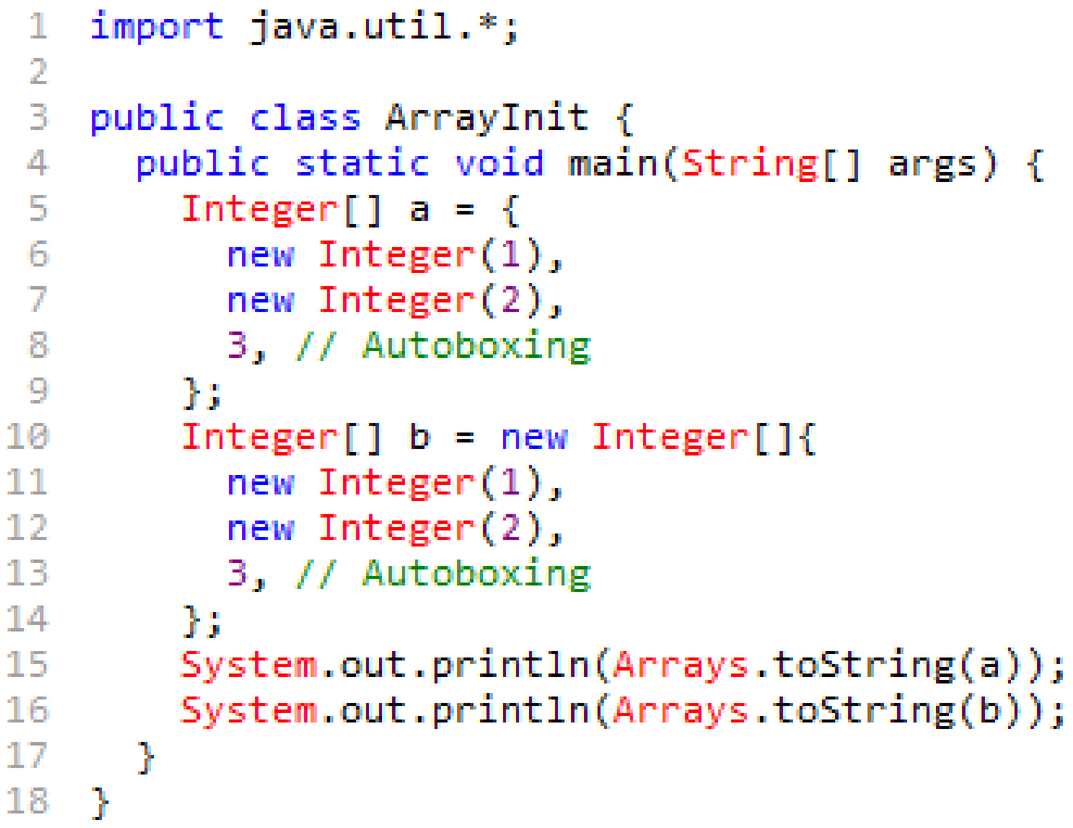
- **If you create a non-primitive array, you create an array of references**
  - ***If you forget to create the object, you’ll get an exception at run time when you try to use the empty array location***



```
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayClassObj {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Integer[] a = new Integer[rand.nextInt(20)];
    print("length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      a[i] = rand.nextInt(500); // Autoboxing
    print(Arrays.toString(a));
  }
}
```

Array initialization – third form

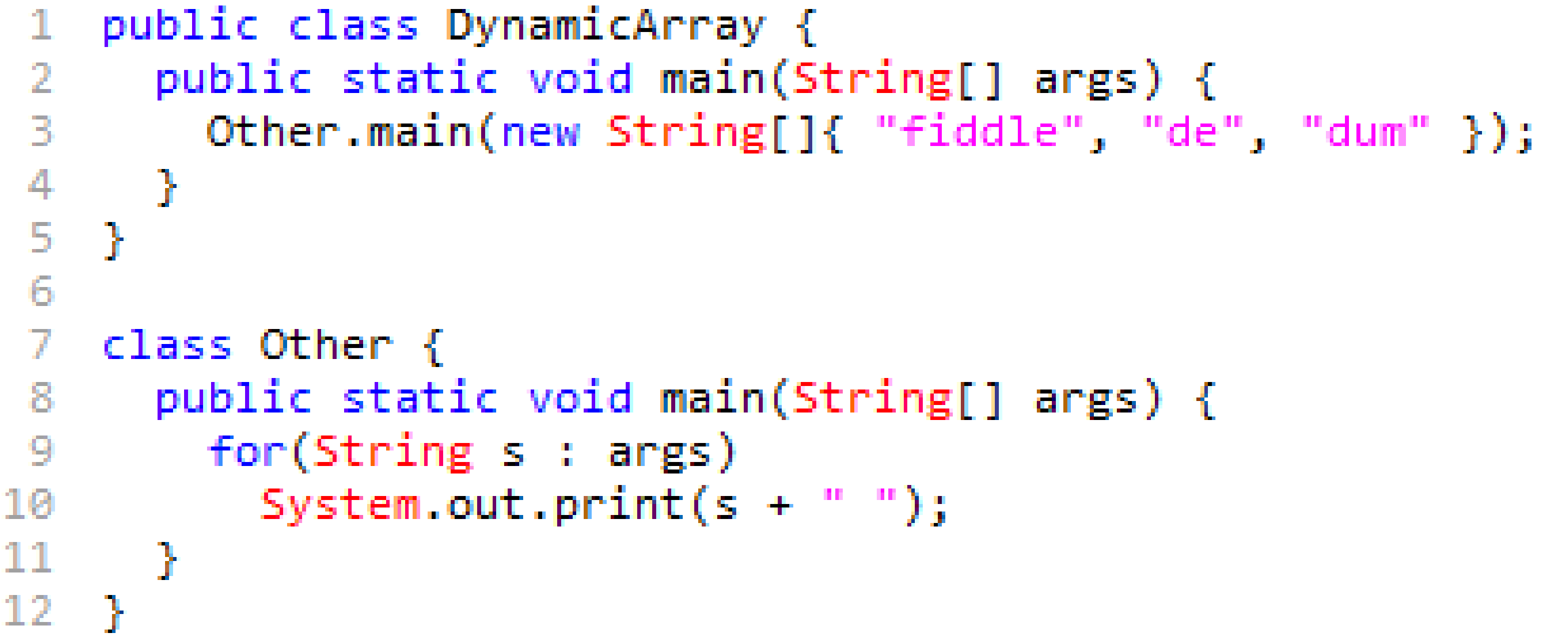
- **Initialize arrays of objects by using the curly brace-enclosed list**
  - ***the final comma in the list of initializers is optional***



```java
import java.util.*;

public class ArrayInit {
  public static void main(String[] args) {
    Integer[] a = {
      new Integer(1),
      new Integer(2),
      3, // Autoboxing
    };
    Integer[] b = new Integer[]{
      new Integer(1),
      new Integer(2),
      3, // Autoboxing
    };
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
  }
}
```

Array initialization

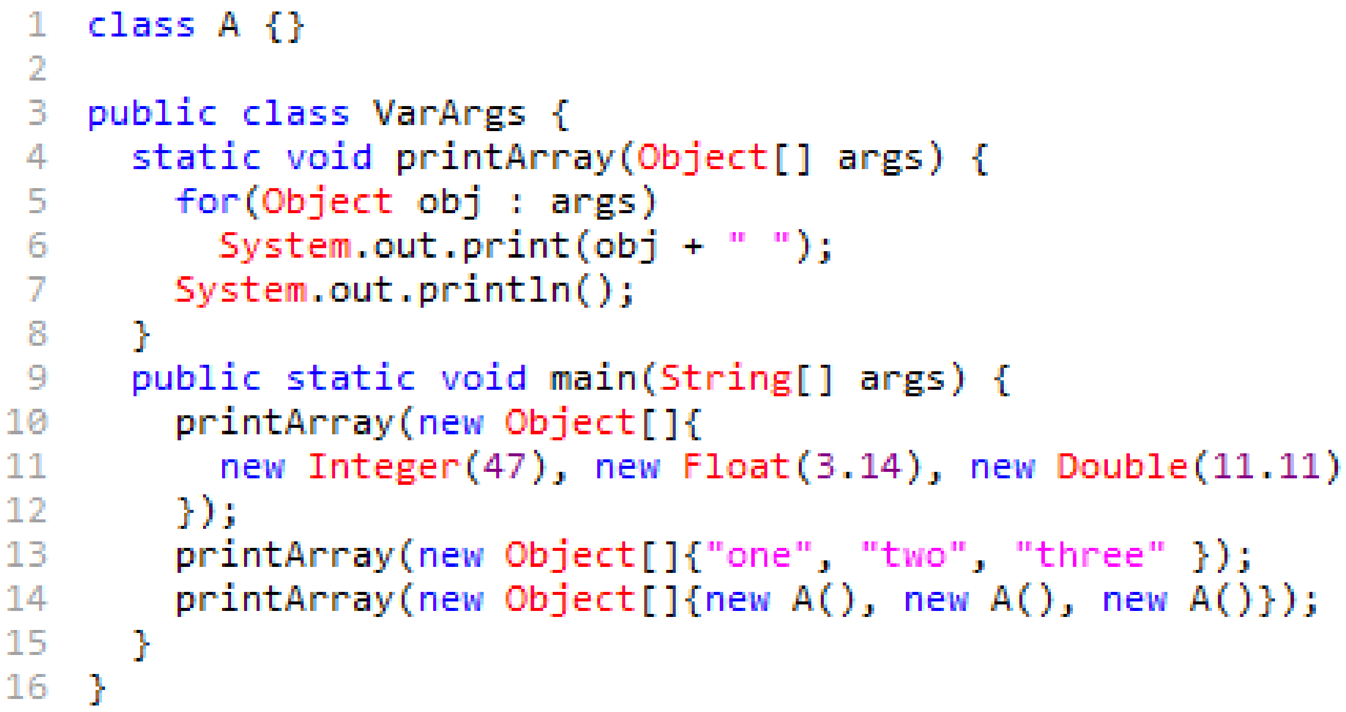
- ❑The first form can only be used at the point where the array is defined
- ❑Use the second and third forms anywhere, even inside a method call



```java
public class DynamicArray {
  public static void main(String[] args) {
    Other.main(new String[]{ "fiddle", "de", "dum" });
  }
}

class Other {
  public static void main(String[] args) {
    for(String s : args)
      System.out.print(s + " ");
  }
}
```

Variable argument lists

- These can include unknown quantities of arguments as well as unknown types

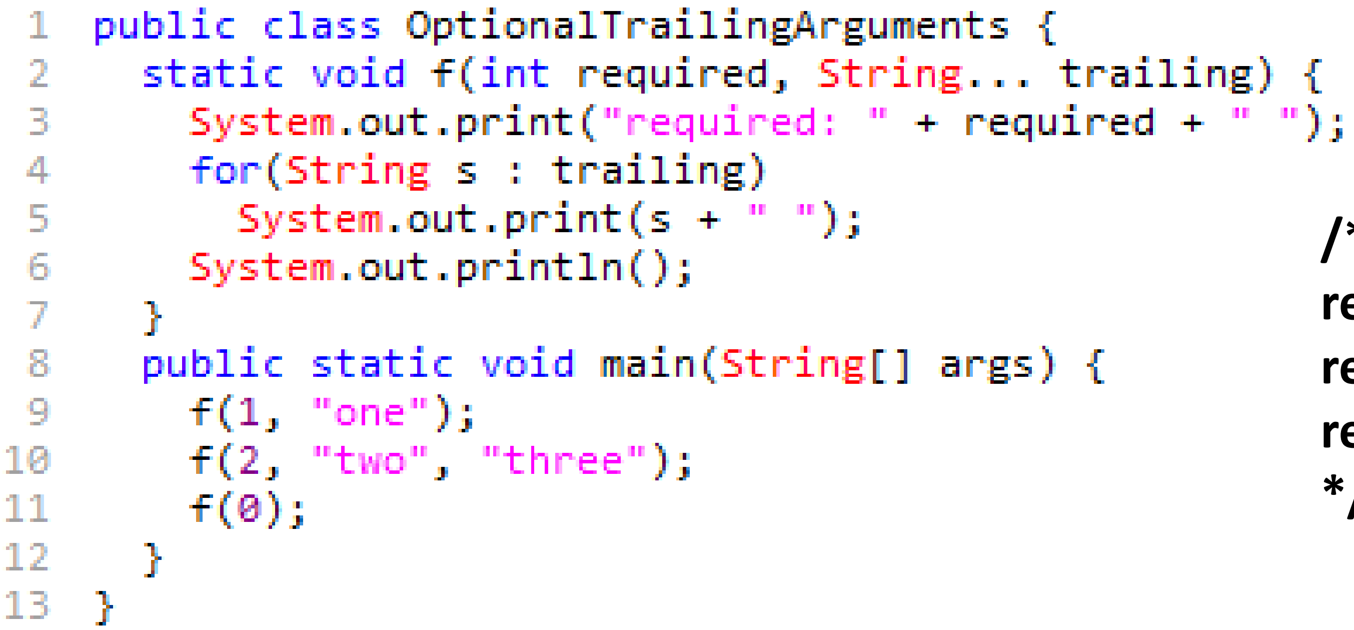


```java
class A {}

public class VarArgs {
  static void printArray(Object[] args) {
    for(Object obj : args)
      System.out.print(obj + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    printArray(new Object[]{
      new Integer(47), new Float(3.14), new Double(11.11)
    });
    printArray(new Object[]{"one", "two", "three" });
    printArray(new Object[]{new A(), new A(), new A()});
  }
}
```

```
/* Output: (Sample)
47 3.14 11.11
one two three
A@1a46e30 A@3e25a5 A@19821f
*///:~
```

Variable argument lists (Cont.)

- With varargs, you no longer have to explicitly write out the array syntax
  - the compiler will actually fill it in for you when you specify varargs
  - it's possible to pass *zero* arguments to a vararg list

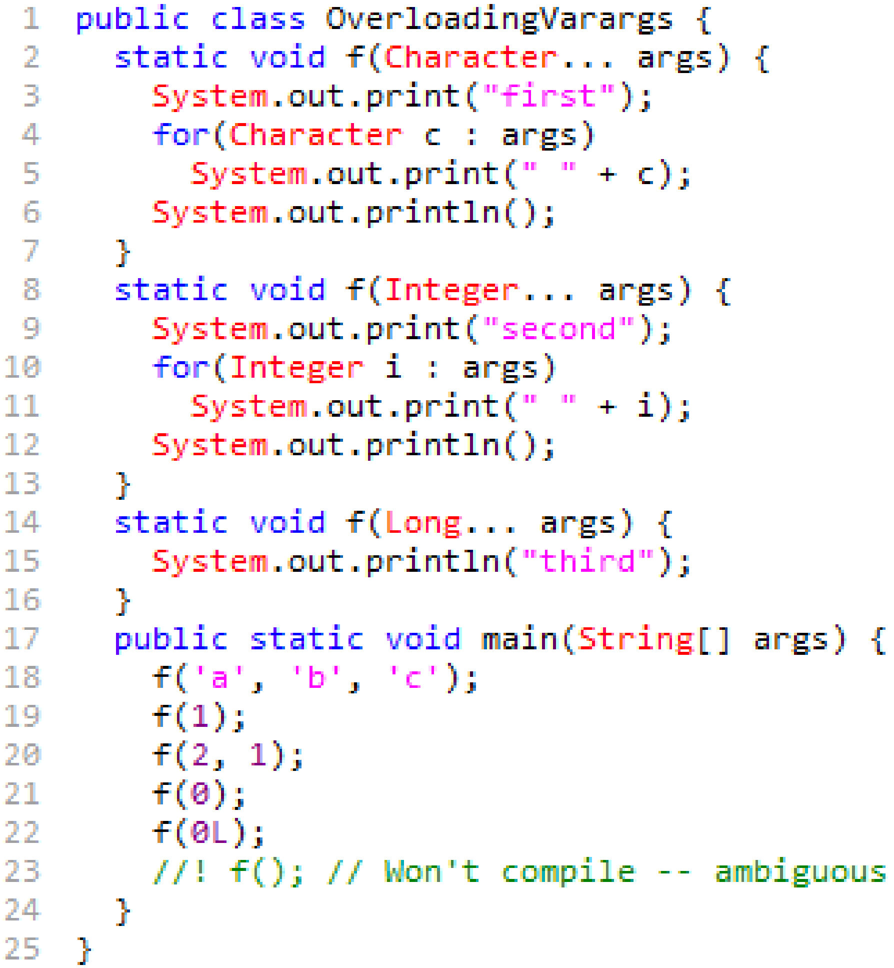


```java
public class OptionalTrailingArguments {
  static void f(int required, String... trailing) {
    System.out.print("required: " + required + " ");
    for(String s : trailing)
      System.out.print(s + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    f(1, "one");
    f(2, "two", "three");
    f(0);
  }
}
```

```
/* Output:
required: 1 one
required: 2 two three
required: 0
*///:~
```

Variable argument lists (Cont.)

- Varargs complicate the process of overloading, although it seems safe enough at first

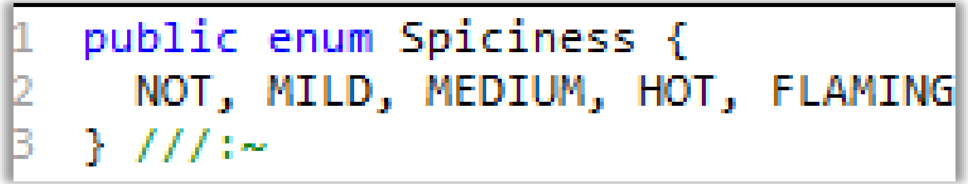


```
public class OverloadingVarargs {
  static void f(Character... args) {
    System.out.print("first");
    for(Character c : args)
      System.out.print(" " + c);
    System.out.println();
  }
  static void f(Integer... args) {
    System.out.print("second");
    for(Integer i : args)
      System.out.print(" " + i);
    System.out.println();
  }
  static void f(Long... args) {
    System.out.println("third");
  }
  public static void main(String[] args) {
    f('a', 'b', 'c');
    f(1);
    f(2, 1);
    f(0);
    f(0L);
    //! f(); // Won't compile -- ambiguous
  }
}
```

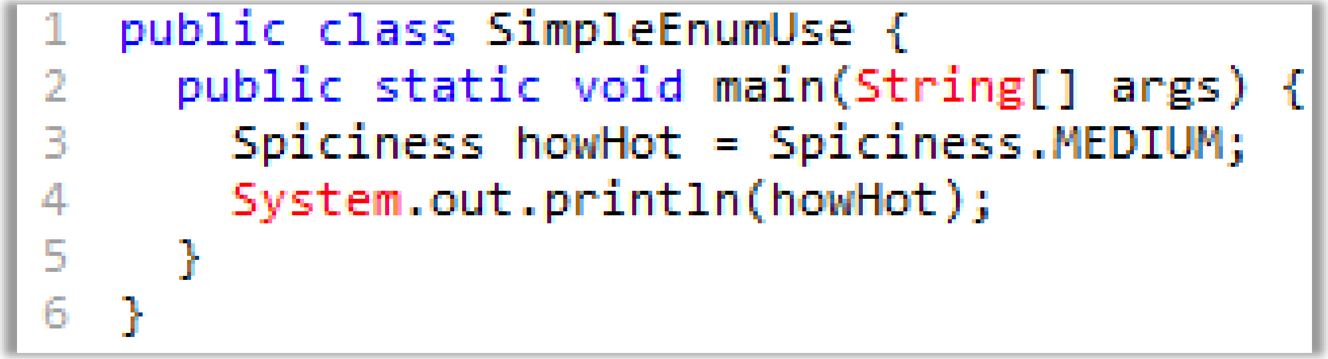
```
/* Output:
first a b c
second 1
second 2 1
second 0
third
*///:~
```

Enumerated types

- **the *enum* keyword**
  - **group together and use a set of *enumerated types***
  - ***enums are classes and have their own methods***



```java
public enum Spiciness {
  NOT, MILD, MEDIUM, HOT, FLAMING
} ///:~
```

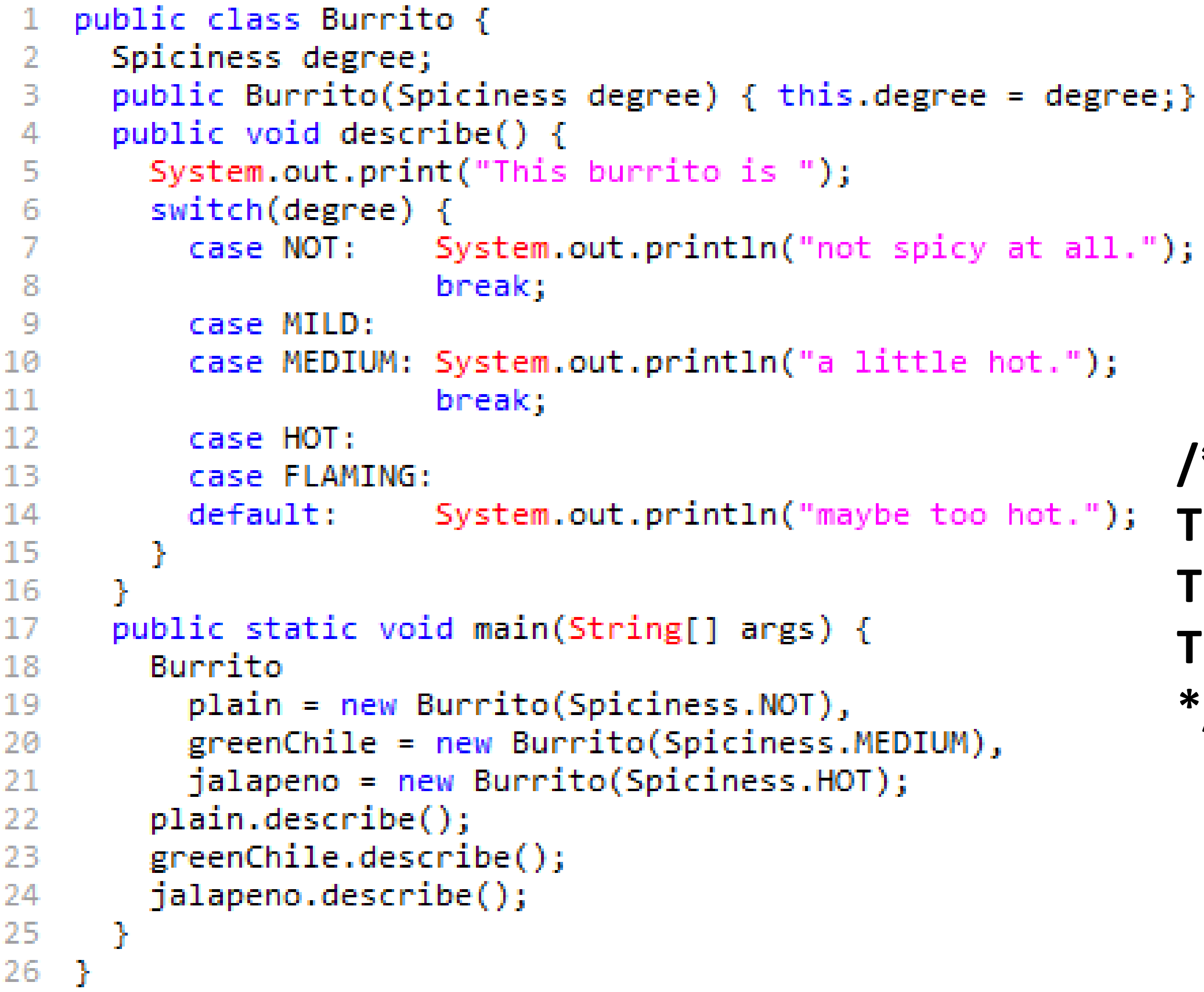


```java
public class SimpleEnumUse {
  public static void main(String[] args) {
    Spiciness howHot = Spiciness.MEDIUM;
    System.out.println(howHot);
  }
}
```

```
/* Output:
MEDIUM
*///:~
```

Enumerated types (Cont.)

- A nice feature is the way that *enums* can be used inside *switch* statements



```
public class Burrito {
  Spiciness degree;
  public Burrito(Spiciness degree) { this.degree = degree;}
  public void describe() {
    System.out.print("This burrito is ");
    switch(degree) {
      case NOT:    System.out.println("not spicy at all.");
                   break;
      case MILD:
      case MEDIUM: System.out.println("a little hot.");
                   break;
      case HOT:
      case FLAMING:
      default:     System.out.println("maybe too hot.");
    }
  }
  public static void main(String[] args) {
    Burrito
      plain = new Burrito(Spiciness.NOT),
      greenChile = new Burrito(Spiciness.MEDIUM),
      jalapeno = new Burrito(Spiciness.HOT);
    plain.describe();
    greenChile.describe();
    jalapeno.describe();
  }
}
```

```
/* Output:
This burrito is not spicy at all.
This burrito is a little hot.
This burrito is maybe too hot.
*///:~
```

Thank you

zhenling@seu.edu.cn