Initialization & Cleanup

Qiuyan Huo
Software Engineering Institute
qyhuo@mail.xidian.edu.cn

Initialization & Cleanup

Java guarantees proper initialization with **constructor**, helps cleanup with **garbage collector**.

Guaranteed initialization with the constructor

- **`initialize()`** – must remember to call it.
- ***constructor***

Same as the class name

First letter is **upper** character

```
class Rock {
    Rock() { // This is the constructor
            System.out.println("Creating Rock");
    }
}
public class SimpleConstructor {
    public static void main(String args[]) {
            for(int i = 0; i < 10; i++)
                    new Rock();
    }
}
```

```
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
Creating Rock
```

constructor

```
class Rock2 {
  Rock2(int i) { // This is the constructor
        System.out.println("Creating Rock number " + i);
  }
}
public class SimpleConstructor2 {
  public static void main(String args[]) {
        for(int i = 0; i < 10; i++)
              new Rock2(i);
  }
}
```

```
Creating Rock number 0
Creating Rock number 1
Creating Rock number 2
Creating Rock number 3
Creating Rock number 4
Creating Rock number 5
Creating Rock number 6
Creating Rock number 7
Creating Rock number 8
Creating Rock number 9
```

Nuance（细微差别）

We can deduce (推断) meaning from context

- “Wash the shirt”
- “Wash the car”
- “Wash the dog”

Not

- “shirtWash the shirt”
- “carWash the car”
- “dogWash the dog”

Method overloading

- void wash (Shirt s) { // …
- void wash (Car c) { // …
- void wash (Dog d) { // …

Unique combinations of argument types distinguish overloaded methods

Method Overloading - *constructor*

- One word, many meanings: *overloaded*

```
class Tree {
    int height;
    Tree() {
      System.out.println("Planting a seedling");
      height = 0;
    }
    Tree(int i) {
      System.out.println(
        "Creating new Tree that is "
        + i + " feet tall");
      height = i;
    }
```

Method Overloading – *common method*

```
    void info() {
        System.out.println("Tree is "
                + height + " feet tall");
    }
    void info(String s) {
        System.out.println(s + ": Tree is "
                + height + " feet tall");
    }
}
```

Method Overloading – cont.

```
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();
  }
}
```

Overloading with primitives

- if you have a data type that is *smaller* than the argument in the method, that data type is promoted
- if your argument is *bigger* than the argument expected by the overloaded method, you must *cast* to the necessary type by placing the type name inside parentheses.
  - If you don't do this, the compiler will issue an error message
  - *narrowing conversion*

Overloading on `return` values?

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

```
void f() {}
int f() {}
```

if you use the call

```
f();
```

  - What would this call mean?

Default constructors: Takes no Arguments

- Compiler automatically creates one for you if you write no constructors

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

```
class Bird {
  int i;
}

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

`this`: Reference to 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();
    }
}
```

`this`: Specifying a Member

- If you get lazy when creating identifiers
- Probably not a good practice, but I do it myself sometimes…

```
class Flower {
  String name;
  Flower(String name) {
        this.name = name;
  }
}
```

Calling constructors from constructors

```
public class Flower {
  int petalCount = 0;
  String s = new String("null");
  Flower(int petals) {
    petalCount = petals;
    System.out.println(
      "Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(String ss) {
    System.out.println(
      "Constructor w/ String arg only, s=" + ss);
    s = ss;
  }
```

```
Flower(String s, int petals) {
    this(petals);//must be the first thing
//!    this(s); // Can't call two!
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
  }
  Flower() {
    this("hi", 47);
    System.out.println("default constructor (no args)");
  }
  void print() {
//! this(11); // Not inside non-constructor!
    System.out.println(
      "petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
  }
}
```

call a constructor from inside non-constructor? NO!

```
Constructor w/ int arg only, petalCount= 47
String & int args
default constructor (no args)
petalCount = 47 s = hi
```

The meaning of `static`

- there is no **`this`** for that particular method
- cannot call non-**`static`** methods from inside **`static`** methods
- you can call a **`static`** method for the class itself, without any object

In fact, this is primarily what a **static** method is for

if you find yourself using a *lot* of static methods, you should probably rethink your strategy

Cleanup: Finalization and Garbage Collection

- Important facts about garbage collection
  - Garbage collection is not destruction
  - Your objects may not get garbage collected
  - Garbage collection is only about memory
- What is **`finalize()`** for?
  - In theory: releasing memory that the GC wouldn't
  - It's never been reliable: promises to be called on system exit; (causes bug in Java file closing)
- You must perform cleanup
  - Must write specific cleanup method

`jvisualvm`

The termination condition

- Using **`finalize()`** to detect an object that hasn't been properly cleaned up.

```
class Book {
  boolean checkedOut = false;
  Book(boolean checkOut) {
    checkedOut = checkOut;
  }
  void checkIn() {
    checkedOut = false;
  }
  public void finalize() {
    if(checkedOut)
      System.out.println("Error: checked out");
  }
}
```

`finalize()` is only useful for obscure memory cleanup that most programmers will never use.

```
public class TerminationCondition {
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Proper cleanup:
    novel.checkIn();
    // Drop the reference, forget to clean up:
    new Book(true);
    // Force garbage collection & finalization:
    System.gc();

  }
}
```

You should do this during program development to speed debugging

Member initialization

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

- Produces compile-time error
- Inside class, primitives are given default values if you don't specify values

```
class Data {
    int i = 999;
    long l; // Defaults to zero
    // ...
}
```

Specifying initialization

- assign the value at the point you define the variable in the class

```
class InitialValues {
  boolean b = true;
  char c = 'x';
  byte B = 47;
  short s = 0xff;
  int i = 999;
  long l = 1;
  float f = 3.14f;
  double d = 3.14159;
  //. . .
```

```
class Measurement {
  Depth d = new Depth();
      boolean b = true;
  // . . .
```

If you haven't given **d** an initial value and you try to use it anyway, you'll get a run-time error called an *exception*

- call a method

```
class CInit {
  int i = f();//g(i)
  //...
}
```

```
class CInit {
  int j = g(i);
  int i = f();
  //...
}
```

Constructor initialization

```
class Counter {
  int i;
  Counter() { i = 7; }
  //……
```

i will first be initialized to **0**, then to **7**

- Order of initialization
  - Order that variables/objects are defined in class

```
class Card {
  Tag t1 = new Tag(1); // Before constructor
  Card() {
    // Indicate we're in the constructor:
    System.out.println("Card()");
    t3 = new Tag(33); // Reinitialize t3
  }
  Tag t2 = new Tag(2); // After constructor
  void f() {
    System.out.println("f()");
  }
  Tag t3 = new Tag(3); // At end
}
```

```
Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

`static` data initialization

you don't initialize

- primitive - standard primitive initial values
- reference to an object - **`null`**

```
class Cupboard {
   Bowl b3 = new Bowl(3);
   static Bowl b4 = new Bowl(4);
   // ...
```

- **`b4`** only created on first access or when first object of class **`Cupboard`** is created

Process of creating an object – class **Dog**

- The first time an object of type **Dog** is created , *or* the first time a **static** method or **static** field of class **Dog** is accessed.
- As **Dog.class** is loaded, all of its **static** initializers are run.
- The construction process for a **Dog** object first allocates enough storage for a **Dog** object on the heap.
- This storage is wiped to zero, automatically setting all the primitives in that **Dog** object to their default values.
- Any initializations that occur at the point of field definition are executed.
- Constructors are executed.

Explicit **static** initialization

```
class Cup {
  Cup(int marker) {
    System.out.println("Cup(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}
class Cups {
  static Cup c1;
  static Cup c2;
  static {
    c1 = new Cup(1);
    c2 = new Cup(2);
  }
  Cups() {
    System.out.println("Cups()");
  }
}
public class ExplicitStatic {
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Cups.c1.f(99);  // (1)
  }
  // static Cups x = new Cups();  // (2)
  // static Cups y = new Cups();  // (2)
}
```

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, is executed only once

```
Inside main()
Cup(1)
Cup(2)
f(99)
```

Non-static instance initialization

```
class Mug {
  Mug(int marker) {
    System.out.println("Mug(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

public class Mugs {
  Mug c1;
  Mug c2;
  {
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    System.out.println("mug1 & mug2 initialized");
  }
  Mugs() {
    System.out.println("Mugs()");
  }
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Mugs m = new Mugs();
  }
}
```

```
Inside main()
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs()
```

Array initialization

```
int a1[];   // This...
int[] a1;   // is the same as this!
```

- Creates a reference, not the array. Can't size it.
- To create an array of primitives:

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

- An array of class objects:

```
Integer[] a = new Integer[pRand(20)];
System.out.println("length of a = " + a.length);
for(int i = 0; i < a.length; i++) {
   a[i] = new Integer(pRand(500));
   System.out.println("a[" + i + "] = " + a[i]);
}
```

- Bounds checked, length produces size

```
length of a = 3
a[0] = 0
a[1] = 0
a[2] = 0
```

Array initialization

- Can also use bracketed list (The size is then fixed at compile-time)

```
Integer[] a = {
    new Integer(1),
    new Integer(2),
    new Integer(3),
};
```

```
int[][] a1 = {
        { 1, 2, 3 },
        { 4, 5, 6 },
    };
```

- If you do anything wrong either the compiler will catch it or an exception will be thrown
- Multi-dimensional arrays shown in book

Variable argument lists

```
Integer[] a = {
    new Integer(1),
    new Integer(2),
    new Integer(3),
};

method(a);
void method(Object...a){
    for(Integer i : a) {
        System.out.print(i + " ")
    }
}
```

Enumerated types

- **enum**

```
public enum Spiciness {
    NOT, MILD, MEDIUM, HOT, FLAMING
}

Spiciness howHot = Spiciness.MEDIUM;
```

Summary of Initialization & Cleanup

- Initialization is critical for objects, thus Java guarantees it with the constructor
- Knowing when to clean up can be difficult in complex systems
- Java GC releases memory only: any other cleanup must be done explicitly!
- Arrays also have Java-style safety