Chapter 4 : Initialization & Cleanup

Two of the safety issues are *initialization* and *cleanup*.

C++ introduced the concept of a *constructor*, a special method automatically called when an object is created. Java also adopted the constructor, and in addition has a garbage collector that automatically releases memory resources when they're no longer being used.

4.1 constructor

In Java, the class designer can guarantee initialization of every object by providing a *constructor*.

The name of the constructor is the same as the name of the class.

```java
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();
  }
} ///:~
```

The constructor can have arguments to specify *how* an object is created.

```java
class Rock2 {
  Rock2(int i) {
    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);
  }
} ///:~
```

The constructor has no return value. This is distinctly different from a **void** return value, in which the method returns nothing but you still have the option to make it return something else. Constructors return nothing and you don’t have an option.

4.2 Method overloading

A method is a name for an action. Well-chosen names make it easier for you and others to understand your code.

Most programming languages (C in particular) require you to have a unique identifier for each function.

In Java (and C++), the constructor forces the overloading of method names.

Method overloading is essential to allow the same method name to be used with different argument types. And although method overloading is a must for constructors, it's a general convenience for any method.

Sample : Overloading.java

(1) Distinguishing overloaded methods

There's a 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.

Sample : OverloadingOrder.java

(2) Overloading with primitives

A primitive can be automatically promoted from a smaller type to a larger one and this can be slightly confusing in combination with overloading.

Sample : PrimitiveOverloading.java

If you have a data type that is smaller than the argument in the method, that data type is promoted.

---- 5→int   char →int

Sample : Demotion.java

The methods take narrower primitive values. If your argument is wider , you must *cast* to the necessary type.

(3) Overloading on return values

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

int x = f ( );

f( );
```

Because you can call a method and ignore the return value, you cannot use return value types to distinguish overloaded methods.

(4) Default constructors

A default constructor is one without arguments. If you create a class with no constructors, the compiler will automatically create a default constructor for you.

If you define any constructors (with or without arguments), the compiler will *not* synthesize a default one for you.

(5) “this”

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

There’s a secret first argument passed to the method **f( )**, and that argument is the reference to the object.

```
Banana.f(a,1);
Banana.f(b,2);
```

For this purpose there's a keyword: **this**. The **this** keyword—which can be used only inside a method—produces the reference to the object the method has been called for.

You can treat this reference just like any other object reference.

Note : If you're calling a method of your class from within another method, you don't need to use **this**; just call the method.

Sample : Leaf.java

Calling constructors from constructors :

Normally, when you say **this**, it is in the sense of "this object".

In a constructor, the **this** keyword takes on a different meaning when you give it an argument list ---- it calls the constructor that matches that argument list. Thus we have a straightforward way to call other constructors. (C++ doesn't allow this action.)

Sample : Flower.java

The constructor **Flower(String s, int petals)** shows that, while you can call one constructor using **this**, you cannot call two. In addition, the constructor call must be the first thing you do.

The compiler won't let you call a constructor from inside any method other than a constructor.

The meaning of static :

It means that there is no **this** for that particular method. You cannot call non-**static** methods from inside **static** methods.

We can call a **static** method for the class itself, without any object.

Putting the **static** method inside a class allows it access to other **static** methods and to **static** fields.

4.3 Cleanup: finalization and garbage collection

Programmers know about the importance of initialization, but often forget the importance of cleanup.

When the garbage collector is ready to release the storage used for your object, it will first call ***finalize( )***, and only on the next garbage-collection pass will it reclaim the object's memory.

Note : In C++ *objects always get destroyed* (in a bug-free program), whereas in Java objects do not always get garbage-collected.

Garbage collection is not destruction :

If there is some activity that must be performed before you no longer need an object, you must perform that activity yourself. ---- finalize( )

Your objects might not get garbage-collected:

If your program completes and the garbage collector never starts to release the storage, that storage will be returned to the operating system as the program exits.

Garbage collection has some overhead. If you never do it you never incur that expense.

(1) What is finalize( ) for?

The garbage collector takes care of the release of all object memory regardless of how the object is created.

The need for **finalize( )** is limited to special cases, in which your object can allocate some storage in some way other than creating an object. ---- native methods

finalize( ) is not the appropriate place for normal cleanup.

(2) Perform cleanup

Java doesn’t allow you to create local objects—you must always use **new**. And because of garbage collection, Java has no destructor. (?)

The garbage collector does not remove the need for or the utility of destructors. (finalize( ) could not be a substitution.)

If you want some kind of cleanup performed other than storage release you must *still* explicitly call an appropriate method in Java, which is the equivalent of a C++ destructor .

One of the things **finalize( )** can be useful for is observing the process of garbage collection.

Sample : Garbage.java

There's a flag called **gcrun** to indicate whether the garbage collector has started running yet. A second flag **f** is a way for **Chair** to tell the **main( )** loop to stop making objects.

The creation of a **String** object during each iteration is simply extra storage being allocated to encourage the garbage collector to run.

If **System.gc( )** is called, then finalization happens to all the objects. Only if **System.gc( )** is called after all the objects are created and discarded will all the finalizers be called.

It seems to make no difference whether **System.runFinalization( )** is called.

Neither garbage collection nor finalization is guaranteed. If the JVM isn't close to running out of memory, it will (wisely) not waste time on garbage collection.

(3) The death condition

There is a very interesting use of **finalize( )** which does not rely on it being called every time. ---- verification of the *death condition* of an object

If one of the finalizations happens to reveal the bug, then you discover the problem.

Sample : DeathCondition.java

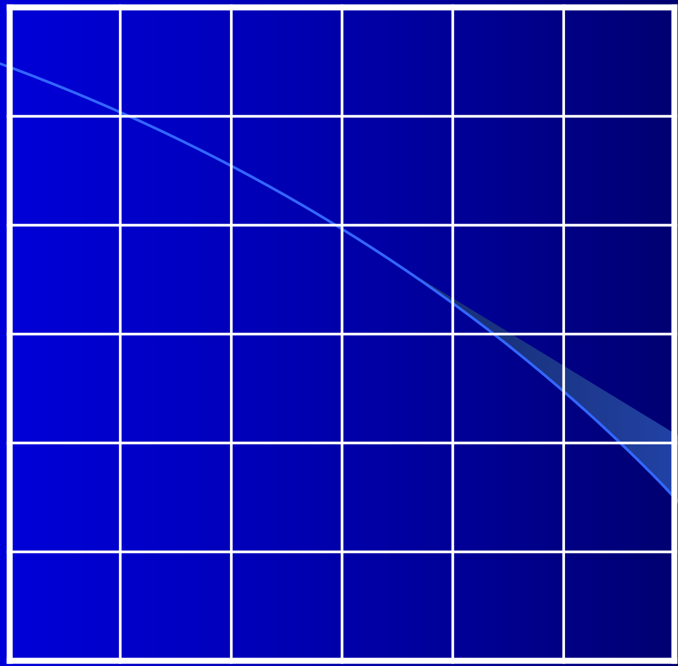
Note : **System.gc( )** is used to force finalization. Even if it isn't, it's highly probable that the bug will eventually be discovered through repeated executions of the program.

(4) How a garbage collector works

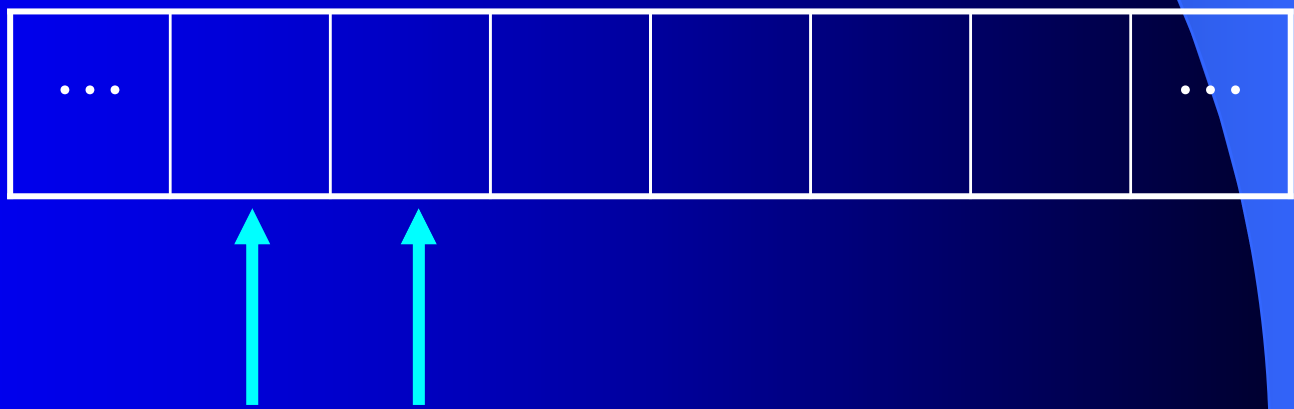
The garbage collector can have a significant impact on *increasing* the speed of object creation. Allocating storage for heap objects in Java can be nearly as fast as creating storage on the stack in other languages.

You can think of the C++ heap as a yard. The Java heap is more like a conveyor belt that moves forward every time you allocate a new object. The "heap pointer" is simply moved forward into blank territory.

C++ heap:



Java heap:



The trick is that the garbage collector steps in and while it collects the garbage it compacts all the objects in the heap. Now the “heap pointer” closer to the beginning of the conveyor belt.

The different garbage collector (GC) schemes :

1. Reference counting

Each object contains a reference counter, and every time a reference is attached to an object the reference count is increased.

The garbage collector moves through the entire list of objects and when it finds one with a zero reference count it releases that storage.

Drawback : if objects circularly refer to each other, they can have nonzero reference counts while still being garbage.

It doesn’t seem to be used in any JVM implementations.

2. stop-and-copy

Any nondead object must ultimately be traceable back to a reference that lives either on the stack or in static storage.

The program is first stopped (not in background). Then, each live object is copied from one heap to another, leaving behind all the garbage. And the new heap is compacted.

when an object is moved from one place to another, all references pointing at it must be changed.

3.mark and sweep

Some JVMs can detect that no new garbage is being generated and switch to a different scheme.

Each time it finds a live object that object is marked by setting a flag. During the sweep, the dead objects are released. However, no copying happens.

Summary :

The Sun garbage collector ran when memory got low.

There are a number of additional speedups possible in a JVM. ---- Loader and Just-In-Time Compiler.

4.4 Member initialization

Java guarantees that variables are properly initialized before they are used.

If variables are defined locally to a method, this guarantee comes in the form of a compile-time error.

```
void f() {
  int i;
  i++;
}
```

Each primitive data member of a class is guaranteed to get an initial value.

Sample : InitialValues.java

When you define an object reference inside a class without initializing it to a new object, that reference is given a special value of **null**.

(1) Specifying initialization

One direct way to do this is simply to assign the value at the point you define the variable in the class. Note : You cannot do this in C++.

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

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

The method can have arguments, but those arguments cannot be other class members that haven't been initialized.

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

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

(2) Constructor initialization

You can call methods and perform actions at run-time to determine the initial values.

Note : The automatic initialization happens before the constructor is entered.

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

Note : i will first be initialized to 0, then to 7. This is true with all the primitive types and with object references, including those that are given explicit initialization at the point of definition.

Order of initialization :

The order of initialization in a class is determined by the order that the variables are defined within the class. All the variables are initialized before any methods can be called.

Sample : OrderOfInitialization.java

Static data initialization :

Placing initialization at the point of definition looks the same as for non-**static**s.

Sample : StaticInitialization.java

The **static** initialization occurs only if it's necessary (when the *first* object is created or the first **static** access occurs). After that, the **static** members are not reinitialized.

The order of initialization is **static**s first, if they haven't already been initialized, and then the non-**static** objects.

Abstract : (class Dog)

1. The first time an object **Dog** is created, *or* the first time a **static** method or **static** field of class **Dog** is accessed, the Java interpreter must locate **Dog.class** .

2. As **Dog.class** is loaded, all of its **static** initializers are run.

3.When you create a **new Dog( )**, the construction process first allocates enough storage on the heap.

4.This storage is wiped to zero, automatically setting all the primitives to their default values and the references to **null**.

5.Any initializations that occur at the point of field definition are executed.

6. Constructors are executed.

Explicit static initialization :

Java allows you to group several **static** initializations inside a special "**static** construction clause".

Sample : ExplicitStatic.java

Non-static instance initialization :

Sample : Mugs.java

(3) Array initialization

Initializing arrays in C is error-prone and tedious.

An array is simply a sequence of either objects or primitives, all the same type and 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. To create storage for the array you must write an initialization expression.

You can use a special kind of initialization expression that must occur at the point where the array is created.

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

Sample : Arrays.java

If you go out of bounds, C and C++ quietly accept this. Java protects you against such problems by causing a run-time error.

Array accesses might be a source of inefficiency in your program.

Creating arrays at run-time :

Sample : ArrayNew.java

The array could have been defined and initialized in the same statement: `int[ ] a = new int [pRand(20)];`

If you're dealing with an array of nonprimitive objects, you must always use **new**. What you create is an array of references.

It's also possible to initialize arrays of objects using the curly-brace-enclosed list.

Sample : ArrayInit.java

variable argument lists :

Sample : VarArgs.java

Multidimensional arrays :

Sample : MultiDimArray.java

```
Integer[ ] [ ] [ ] a5 = new Integer[3][2][4];
a5[0][0][1] = new Integer(15);
```