Chapter 9
Classes and Objects: A Deeper Look (Ⅱ)

TAO Yida

taoyd@sustech.edu.cn

Objectives

- To use `static` variables and methods
- Declare constants with the `final` keyword
- To organize classes in packages to promote reuse
- Class member access levels
- Enumerations
- Stack and heap memory

`static` Class Members

- Recall that every object of a class has its own copy of all the instance variables of the class.
  - Instance variables represent concepts that are unique per instance, e.g., `name` in class `Student`.
- In certain cases, only one copy of a particular variable should be shared by all objects of a class (e.g., a counter that keeps track of every object created for memory management).
  - A `static` field—called a class variable—is used in such cases.

`static` Class Members

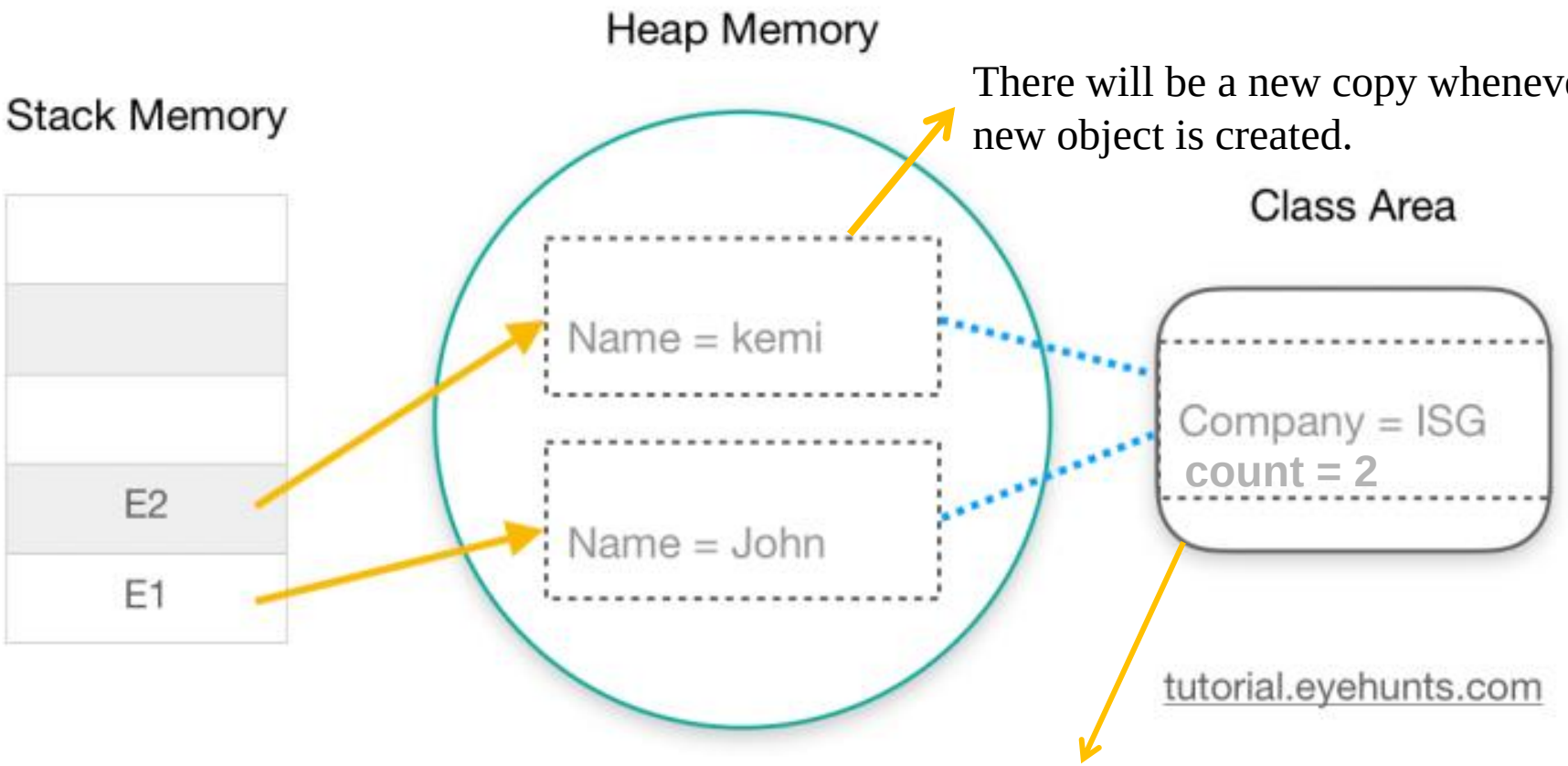
- A `static` variable represents classwide information. All objects of the class share the same piece of data.

```
public class Employee {

    private String firstName;

    private String lastName;

    private static int count; // number of employees created

}
```

There will be a new copy whenever a new object is created.

There is only one copy for each static variable. Make a variable `static` when all objects of the class must use the same copy of the variable.

`static` Class Members



There is only one copy for each static variable. Make a variable `static` when all objects of the class must use the same copy of the variable.

`static` Class Members

- `static` class members are available as soon as the class is loaded into memory at execution time (objects may not exist yet)
- A class's `public static` members can be accessed through a reference to any object of the class, or by qualifying the member name with the class name and a dot (.), e.g., `Math.PI`

```
public class EmployeeTest { ...
  public static void main(String[] args) {
    Employee e = new Employee();
    System.out.printf("# employees = %d", e.count); // not encouraged
    System.out.printf("# employees = %d", Employee.count); // good practice
  }
}
```

`static` Class Members

- A class's `private static` members can be accessed by client code only through methods of the class

```
public class Employee {
    private String firstName;
    private String lastName;
    private static int count; // number of employees created
    public static int getCount() {  return count;  }
}
public class EmployeeTest {
    public static void main(String[] args) {
        System.out.printf("# employees = %d", Employee.getCount());
    }
}
```

`static` Class Members

- A `static` method cannot access non-static class members (e.g., instance variables), because a static method can be called even when no objects of the class have been instantiated.

- For the same reason, the `this` reference cannot be used in a `static` method.

- If a `static` variable is not initialized, the compiler assigns it a default value (e.g., 0 for `int`)

Example

```
public class Employee {
    private String firstName;
    private String lastName;
    private static int count; // number of employees created

    public Employee(String first, String last) {
        firstName = first;
        lastName = last;
        ++count;
        System.out.printf("Employee constructor: %s %s; count = %d\n",
                          firstName, lastName, count);
    }

    public String getFirstName() {  return firstName;  }

    public String getLastName() {  return lastName;  }

    public static int getCount() {  return count;  }
}
```

Example

```
public class EmployeeTest {
  public static void main(String[] args) {
    System.out.printf("Employees before instantiation: %d\n",
                      Employee.getCount());
    Employee e1 = new Employee("Bob", "Blue");
    Employee e2 = new Employee("Susan", "Baker");
    System.out.println("\nEmployees after instantiation:");
    System.out.printf("via e1.getCount(): %d\n", e1.getCount());
    System.out.printf("via e2.getCount(): %d\n", e2.getCount());
    System.out.printf("via Employee.getCount(): %d\n", Employee.getCount());
    System.out.printf("\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
                      e1.getFirstName(), e1.getLastName(),
                      e2.getFirstName(), e2.getLastName());
  }
}
```

The only way to access static variables at this stage

More choices when there are objects

Example

```
Employees before instantiation: 0
Employee constructor: Bob Blue; count = 1
Employee constructor: Susan Baker; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Bob Blue
Employee 2: Susan Baker
```

Access the same variable

Objectives

- To use `static` variables and methods
- Declare constants with the `final` keyword
- To organize classes in packages to promote reuse
- Class member access levels
- Enumerations
- Stack and heap memory

`final` Instance Variables

- The principle of least privilege (PoLP, 最小特权原则) is fundamental to good software engineering
  - Code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more (any user, program, or process should have only the bare minimum privileges necessary to perform its function).
  - Makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.

`final` Instance Variables

- The keyword `final` specifies that a variable is not modifiable (i.e., constant) and any attempt to modify leads to an error (cannot compile)

```
private final int INCREMENT;
```

- Generally, every field in an object or class is initialized to a zero-like value during the allocation of memory (primitive types start out with zero values, object types start out as null, and Boolean types start out false).

- However, there is an exception to this behavior for `final` fields, which are required to be explicitly initialized. If this is not done, the code will fail to compile.

`final` Instance Variables

- Two ways to initialize a final variable
  - `final` variables can be initialized when they are declared.

    ```
    public static final double PI = 3.141592653589793238462;
    ```

  - If they are not, they must be initialized in every constructor of the class (Initializing `final` variables in constructors enables each object of the class to have a different value for the constant)
- If a `final` variable is not initialized when it is declared or in every constructor, the program will not compile.

Objectives

- To use `static` variables and methods
- Declare constants with the `final` keyword
- To organize classes in packages to promote reuse
- Class member access levels
- Enumerations
- Stack and heap memory

Creating Packages

- Each class in the Java API belongs to a package that contains a group of related classes.
- Packages help programmers organize application components (logically related classes can be put into the same package (e.g., java.io)).
- Packages facilitate software reuse by enabling programs to import classes from other packages, rather than copying the classes into each program that uses them.

Declaring a reusable class

- **Step 1:** Declare a `public` class (to be reusable)
- **Step 2:** Choose a package name and add a `package` declaration to the source file for the reusable class declaration.
  - In each Java source file there can be only one `package` declaration, and it must precede all other declarations and statements.

```
package sustech.cs102a;

public class Time {
    private int hour; // 0 - 23
    private int minute; // 0 - 59
    private int second; // 0 - 59
    //...
}
```

Creating Packages (Cont.)

- A Java source file must have the following order:
  - a `package` declaration (if any)
  - `import` declarations (if any)
  - class declarations (you can declare multiple classes in one `.java` file)
    - only one of the class declarations in a `.java` file can be `public`.
    - Other classes in the file are placed in the package and can be used only by the other classes in the package (package-private). Non-`public` classes are in a package to support the reusable classes in the package.
    - Think: Can classes be declared as private?

Creating Packages (Cont.)

- A Java package structure is like a directory structure. Its a tree of packages, subpackages and classes inside these classes. A Java package structure is indeed organized as directories on your hard drive, or as directories inside a zip file (JAR files)

- The class `Time` should be placed in the directory

```
sustech
    cs102a
```

```
package sustech.cs102a;

public class Time {
    private int hour; // 0 - 23
    private int minute; // 0 - 59
    private int second; // 0 - 59
    //...
}
```

Creating Packages (Cont.)

- `javac` command-line option `-d` causes the compiler to create appropriate directories based on the class's `package` declaration.

> **-d *directory***
>
> Set the destination directory for class files. The directory must already exist; **javac** will not create it. If a class is part of a package, **javac** puts the class file in a subdirectory reflecting the package name, creating directories as needed. For example, if you specify **-d C:\myclasses** and the class is called `com.mypackage.MyClass`, then the class file is called `C:\myclasses\com\mypackage\MyClass.class`.
>
> If **-d** is not specified, **javac** puts each class files in the same directory as the source file from which it was generated.

- Example command: `javac -d . Time.java`
  - specifies that the first directory in our package name should be placed in the current directory (`.`)
  - The compiled classes are placed into the directory that is named last in the `package` declaration
  - `Time.class` will appear in the diretroy `./sustech/cs102a/`

Creating Packages (Cont.)

- `package` name is part of the fully qualified name of a class
  - `sustech.cs102a.Time`
- We can use the fully qualified name in programs, or `import` the class and use its simple name (e.g., `Time`).
- If another package contains a class of the same name, the fully qualified class names can be used to distinguish between the classes in the program and prevent a name conflict

Importing a class

- A single-type-import declaration specifies one class to import
  - `import java.util.Scanner;`
- When your program uses multiple classes from the same package, you can import them with a type-import-on-demand declaration.
  - `import java.util.*; // import java.util classes`
- The wild card * informs the compiler that all `public` classes from the `java.util` package are available for use in the program.

`static` Import

- Normal import declarations import classes from packages, allowing them to be used without package qualification
- A `static` import declaration enables you to import the `static` members (fields or methods) of a class so you can access them via their unqualified names, i.e., without including class name and a dot (`.`)
  - `Math.sqrt(4.0)` → `sqrt(4.0)`

Example

```
1  // Fig. 8.14: StaticImportTest.java
2  // Static import of Math class methods.
3  import static java.lang.Math.*;
4
5  public class StaticImportTest
6  {
7     public static void main( String[] args )
8     {
9        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
10       System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
11       System.out.printf( "log( E ) = %.1f\n", log( E ) );
12       System.out.printf( "cos( 0.0 ) = %.1f\n", cos( 0.0 ) );
13    } // end main
14 } // end class StaticImportTest
```

Enables `Math` methods to be used by their simple names in this file

```
sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
log( E ) = 1.0
cos( 0.0 ) = 1.0
```

Ambiguity in static import

- If two static members of the same name are imported from multiple different classes, the compiler will throw an error, as it will not be able to determine which member to use in the absence of class name qualification.

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;

public class Demo {
    public static void main(String[] args) {
        System.out.println(MAX_VALUE);
    }
}
```

Reference to 'MAX_VALUE' is ambiguous, both 'Integer.MAX_VALUE' and 'Byte.MAX_VALUE' match

Import static constant... Alt+Shift+Enter More actions... Alt+Enter

No documentation found.

Specifying Classpath (Compile-time)

- When compiling a class that uses classes from other packages, `javac` must locate the `.class` files for all these classes.
- The compiler locate the classes as follows:
  - It begins by searching the standard Java classes that are bundled with the JDK (e.g., all classes in `java.lang`).
  - Then it searches for the current directory by default (.)
  - The classpath can be modified by
    - providing the `-classpath (-cp)` option to the `javac` compiler (override the default current directory)
    - setting the `CLASSPATH` environment variable (not recommended).

Specifying Classpath (Compile-time)

- The classpath consists of a list of directories or archive files, each separated by a directory separator
  - Semicolon (`;`) on Windows, colon (`:`) on UNIX/Linux/Mac OS X

```
javac -classpath .:/home/avh/classes:/usr/local/java/classes Test.java
```

- Archive files are individual files that contain directories of other files, typically in a compressed format
  - Normally end with the `.jar` or `.zip` file-name extensions

- The directories and archive files specified in the classpath contain the classes you wish to make available to the compiler and the JVM

Specifying Classpath (Runtime)

- When you execute an application, the JVM must be able to load the `.class` files of the classes used in that application.
- The `java` command uses a class loader that searches
  - Bootstrap classes: core Java classes (rt.jar)
  - Extension classes: .jar in %JAVA_HOME%\jre\lib\ext
  - User classes: specified by the classpath (the current directory by default).
- The classpath can be specified explicitly by using either of the techniques discussed for the compiler.

```
java -classpath .:/home/avh/classes:/usr/local/java/classes Test
```

Compile-time classpath vs Runtime classpath

- You compile the code with a given library on the compile-time classpath, but forget to add it to the runtime classpath. The JVM throws NoClassDefFoundError

- Your code has new Clazz(), but the statement is never really executed during runtime (if condition not met); in this case, you need to find Clazz.class in compile-time classpath, but no need to find it in runtime classpath

- Your code runs in a Framework (Spring); your code doesn't need json.jar, but the framework needs it internally. In this case, you don't need to add this jar in compile-time classpath, but need to add it to runtime classpath

Objectives

- To use `static` variables and methods
- Declare constants with the `final` keyword
- To organize classes in packages to promote reuse
- Class member access levels
- Enumerations
- Stack and heap memory

Package Access

- If no access modifier is specified for a class member when it's declared in a class, it is considered to have package access.

```
public class Time1 {
    int hour;
    int minute;
    int second;
    void setTime(int h, int m, int s) {...}
}
```

No modifier

The variables and method are package-private, visible only to classes of the same package

Access Level Modifiers (So Far)

Modifier	Class	Package	World
public	Y	Y	Y
no modifier	Y	Y	N
private	Y	N	N

Note that this is for controlling access to class members. At the top level, a class can only be declared as `public` or package-private (no explicit modifier)

Example: Package Access

```
// class with package access instance variables
class PackageData
{
   int number; // package-access instance variable
   String string; // package-access instance variable

   // constructor
   public PackageData()
   {
      number = 0;
      string = "Hello";
   } // end PackageData constructor

   // return PackageData object String representation
   public String toString()
   {
      return String.format( "number: %d; string: %s", number, string );
   } // end method toString
} // end class PackageData
```

Class has package access; can be used only by other classes in the same directory

Package access data can be accessed by other classes in the same package via a reference to an object of the class

Example: Package Access

```
public class PackageDataTest
{
   public static void main( String[] args )
   {
      PackageData packageData = new PackageData();

      // output String representation of packageData
      System.out.printf( "After instantiation:\n%s\n", packageData );

      // change package access data in packageData object
      packageData.number = 77;
      packageData.string = "Goodbye";

      // output String representation of packageData
      System.out.printf( "\nAfter changing values:\n%s\n", packageData );
   } // end main
} // end class PackageDataTest
```

```
After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye
```

Accessing package access variables in class PackageData

Package access is rarely used in practice.

Objectives

- To use `static` variables and methods
- Declare constants with the `final` keyword
- To organize classes in packages to promote reuse
- Class member access levels
- Enumerations
- Stack and heap memory

Enumerations

- There are cases when a variable can only take one of a small set of predefined constant values, e.g., compass direction (N, S, E, W) and the days of a week (MON, TUE, etc.)
- In such cases, you should use an `enum` type to define a set of constants represented as unique identifiers

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST
}
```

Enumerations

- `Direction` is a type called an **enumeration**, which is a special kind of class introduced by the keyword `enum` and a type name
- Inside the braces {} is a comma-separated list of enumeration constants, each representing a unique value (you don't need to care about the underlying implementation or the exact values)
- The identifiers in an `enum` must be unique

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST
}
```

Enumerations

- Variables of the type `Direction` can be assigned only the four constants declared in the enumeration (other values are illegal, won't compile)
  - `Direction d = Direction.NORTH;`
- Like classes, all `enum` types are reference types

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST
}
```

Enumerations

- Each `enum` declaration declares an `enum` class with the following restrictions:
  - `enum` constants are implicitly `final` (constants that shouldn't be modified)
  - `enum` constants are implicitly `static` (no objects need to access them)
  - `enum` declarations contain two parts: (1) the `enum` constants, (2) the other members such as constructor, fields and methods (optional)
  - An `enum` constructor can specify any number of parameters and can be overloaded
  - An enum constructor cannot be public; Any attempt to create an object of an `enum` type with operator `new` results in a compilation error (If no access modifier is specified for the constructor of an enum type, the constructor is private.)

Example

enum constants (objects in this example) initialized with constructor calls

```
public enum Book {
    JHTP("Java How to Program", "2012"),
    CHTP("C How to Program", "2007"),
    IW3HTP("Internet & World Wide Web How to Program", "2008"),
    CPPHTP("C++ How to Program", "2012"),
    VBHTP("Visual Basic 2010 How to Program", "2011"),
    CSHARPHTP("Visual C# 2010 How to Program", "2011");

    private final String title;
    private final String copyrightYear;
    private Book(String bookTitle, String year) {
        title = bookTitle;
        copyrightYear = year;
    }
    public String getTitle() {  return title;  }
    public String getCopyrightYear() {  return copyrightYear;  }
}
```

Just like normal classes, defining public methods for clients to use the enum type

Only six Book objects will be created, constants such as Book.JHTP store the references.

Enumerations

- For every `enum`, the compiler generates the `static` method `values` that returns an array of the `enum`'s constants.
- When an `enum` constant is converted to a `String`, the constant's identifier is used as the `String` representation.

```
public static void main(String[] ars){
    for (Direction d : Direction.values()) {
        System.out.println(d.toString());
    }
}
```

```
NORTH
SOUTH
EAST
WEST
```

Objectives

- To use `static` variables and methods
- Declare constants with the `final` keyword
- To organize classes in packages to promote reuse
- Class member access levels
- Enumerations
- Stack and heap memory

Java Heap Memory

- The heap space is used by Java runtime to allocate memory to Objects and JRE classes. Whenever we create an object (including arrays), it's created in the heap space.

- Any object created in the heap space has global access and can be referenced from anywhere of the application (as long as you have a reference)

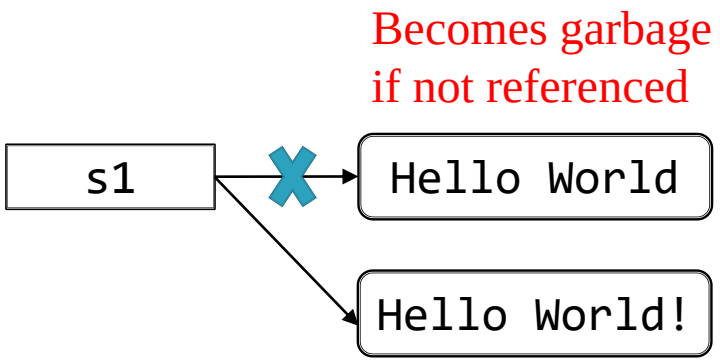
- Garbage Collection runs on the heap memory to free the memory used by objects that doesn't have any reference.

https://www.journaldev.com/4098/java-heap-space-vs-stack-memory

Garbage Collection

- Every object uses system resources, such as memory
- We need a disciplined way to give resources back to the system when they're no longer needed; otherwise, resource leaks may occur.
- The JVM performs automatic garbage collection to reclaim the memory occupied by objects that are no longer used (no references to them).

```
String s1 = "Hello World";
s1 = s1.concat("!");
```



Garbage Collection

- With garbage collection, memory leaks that are common in other languages like C and C++ (memory is not automatically reclaimed in those languages) are less likely in Java, but some can still happen in subtle ways.

- Other types of resource leaks can occur
  - An application may open a file on disk to modify its contents.
  - If it does not close the file, the application must terminate before any other application can use it (here the file is exclusive resource).

Java Stack Memory

- Stack memory stores information for execution of methods in a thread:
  - Method specific values (short-lived)
  - References to other objects in the heap (getting referred from the methods)
- Stack memory is always referenced in LIFO order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and references to other objects.
- As soon as a method ends, the block will be erased and become available for next method. Therefore, stack memory size is very less compared to heap memory (storing long-lived objects).

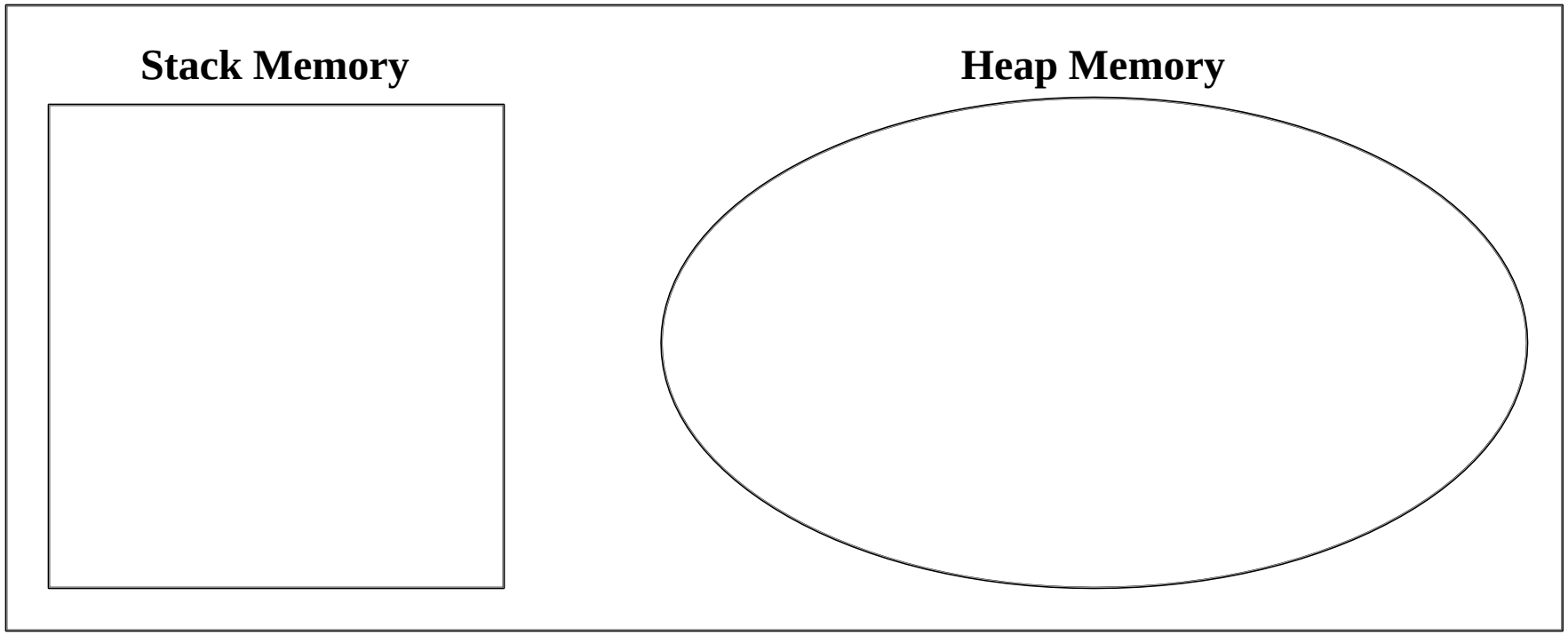
https://www.journaldev.com/4098/java-heap-space-vs-stack-memory

Memory Allocation Example

```
public class Memory {

    public static void main(String[] args) {
        int i = 1;
        Object obj = new Object();
        Memory mem = new Memory();
        mem.foo(obj);
    }

    private void foo(Object param) {
        String str = param.toString();
        System.out.println(str);
    }

}
```

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

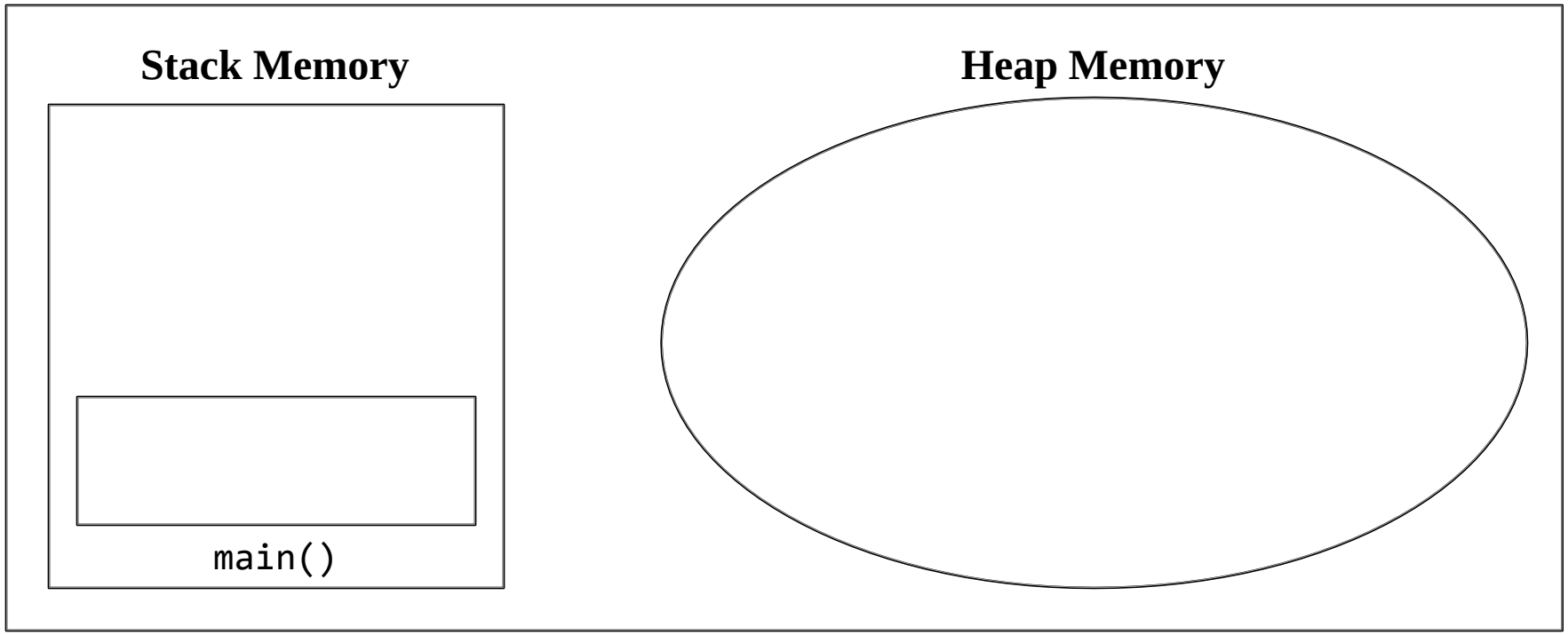
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

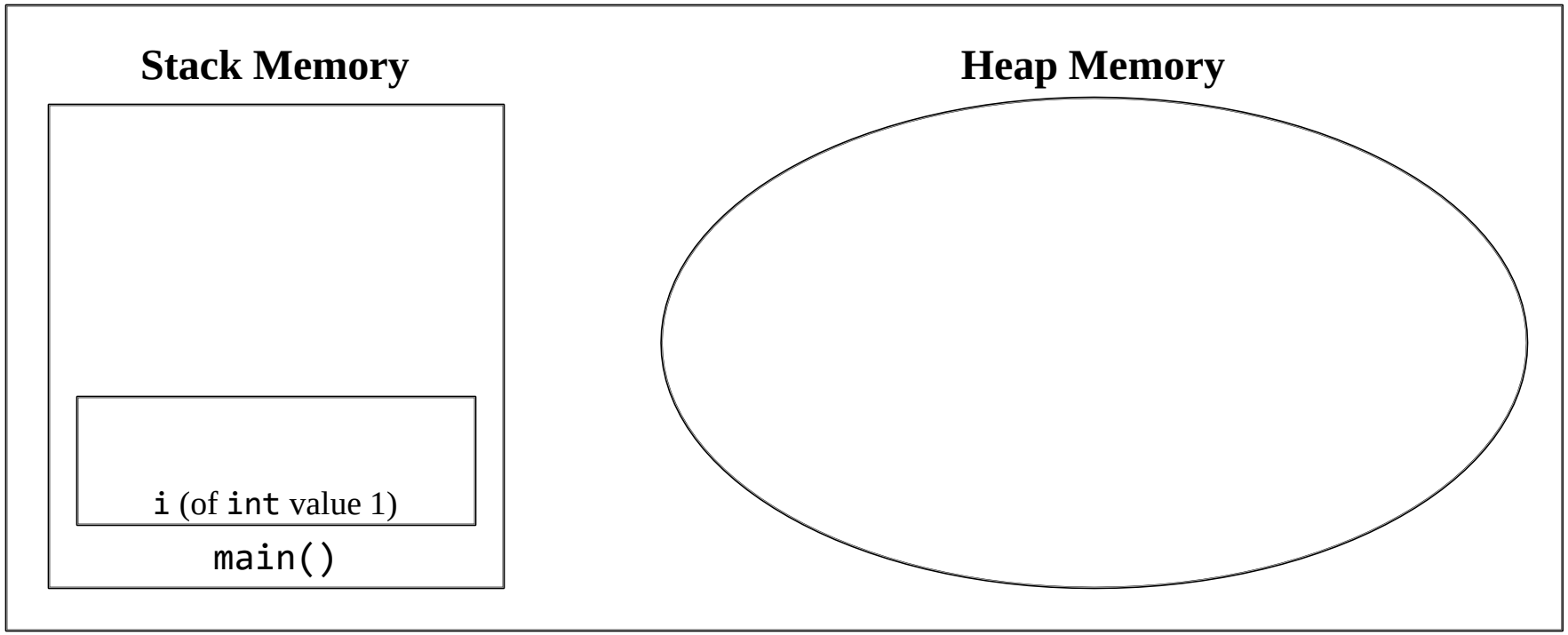
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

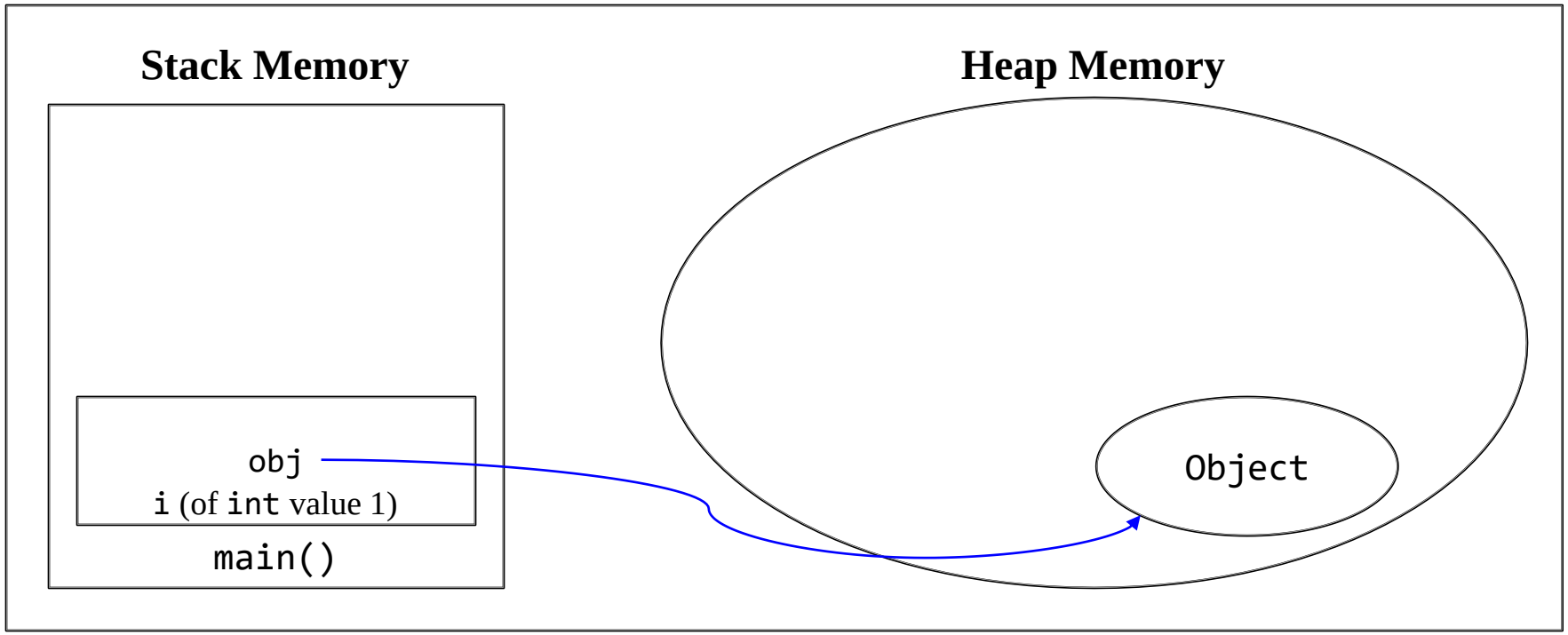
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

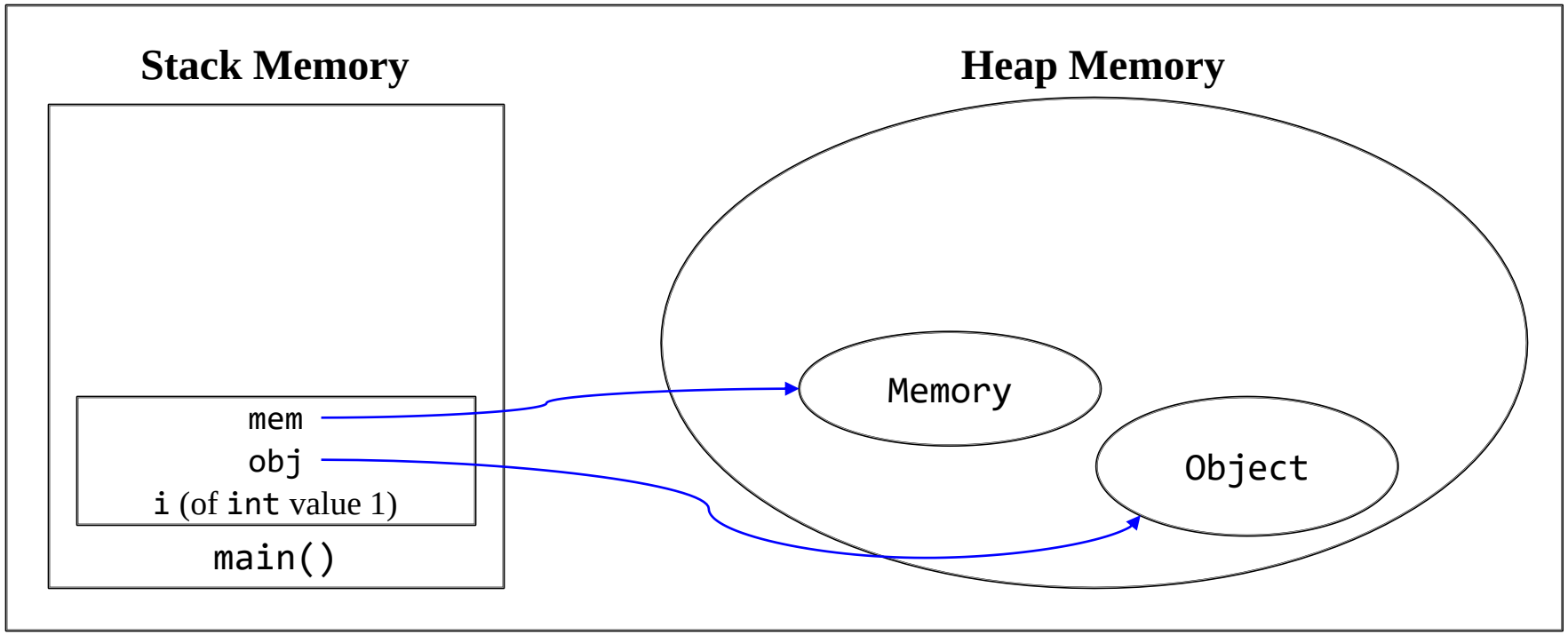
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

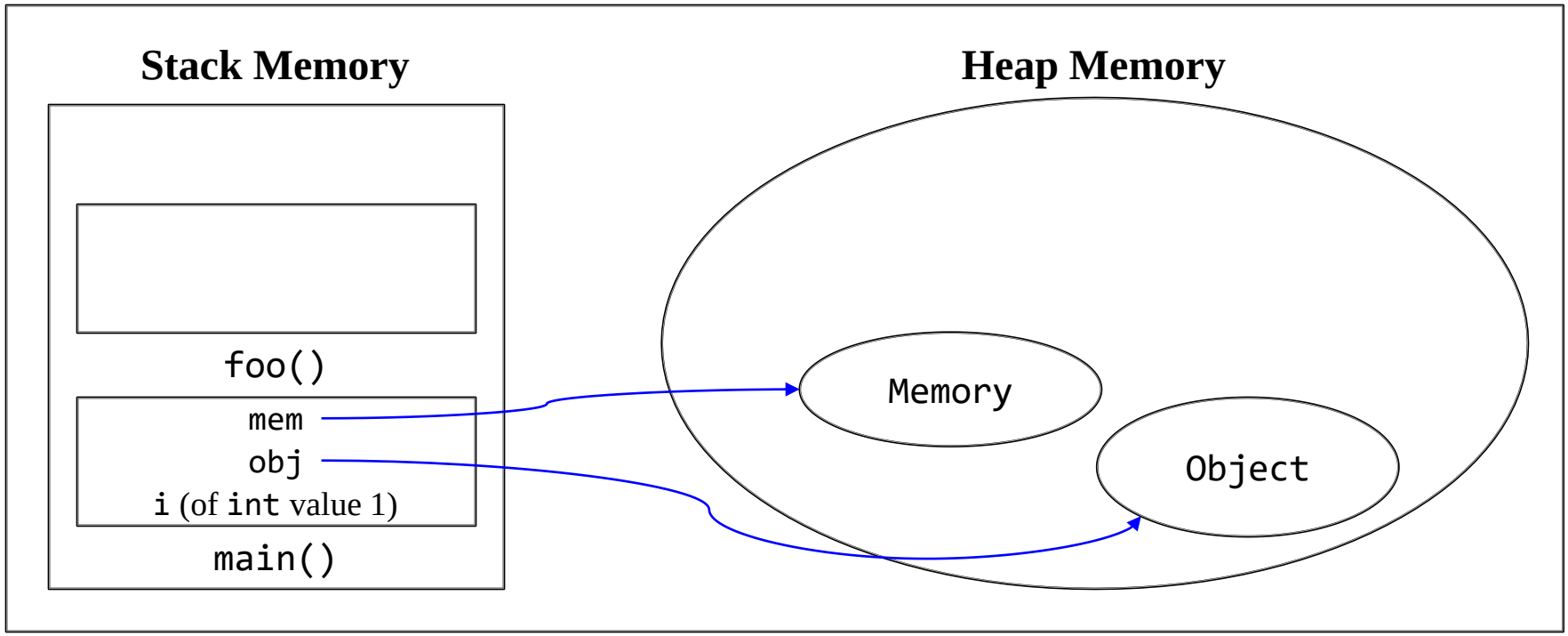
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

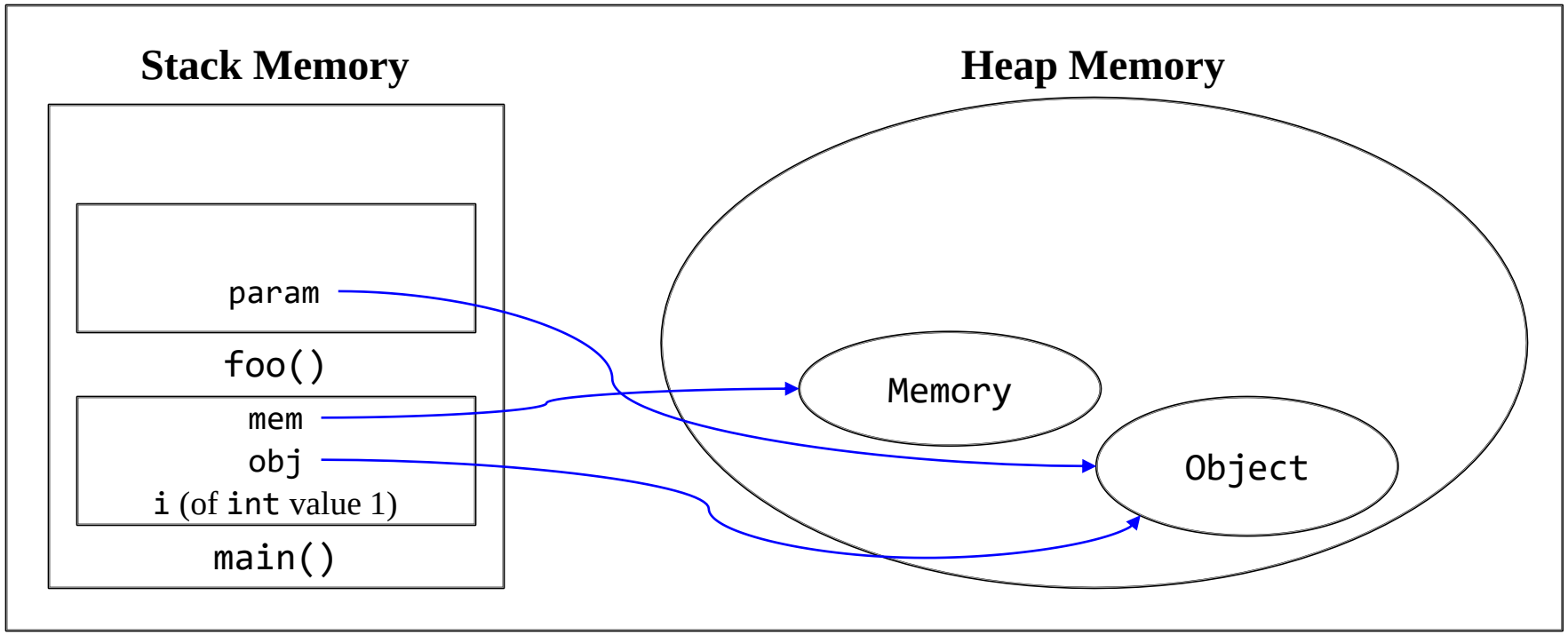
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

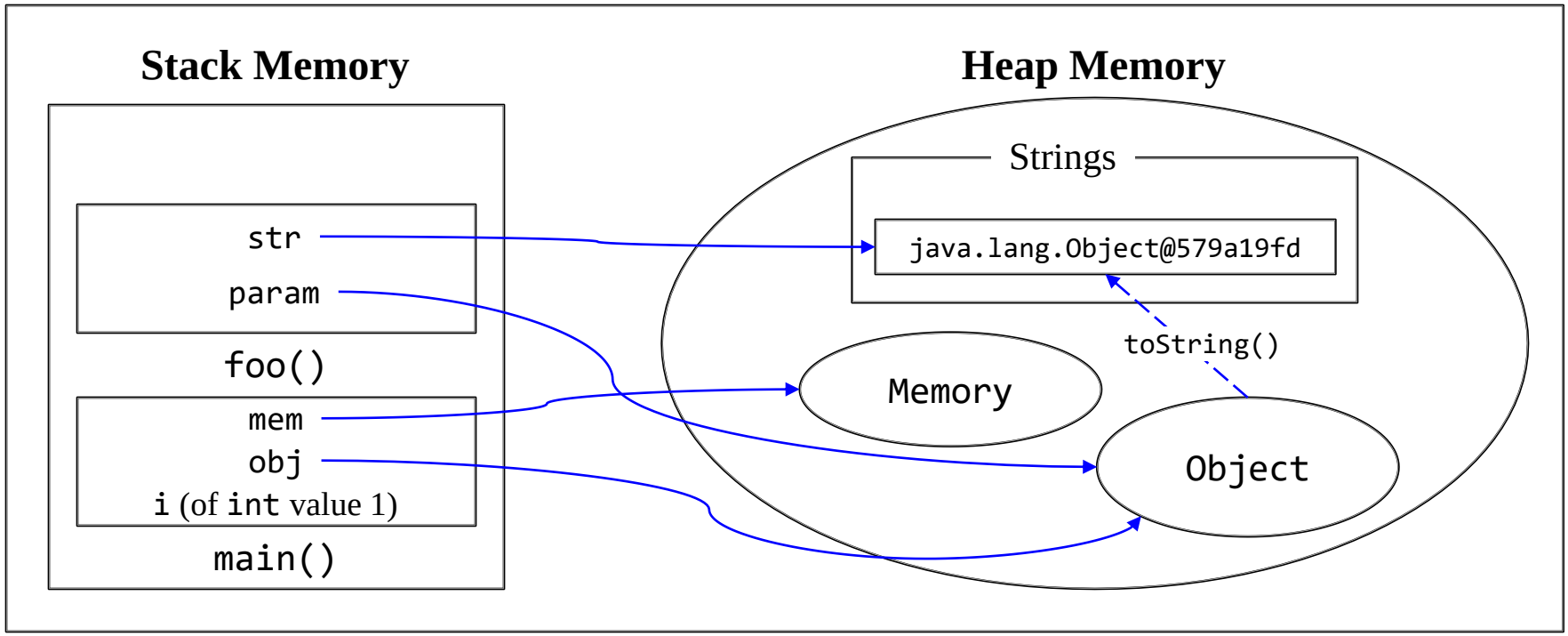
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

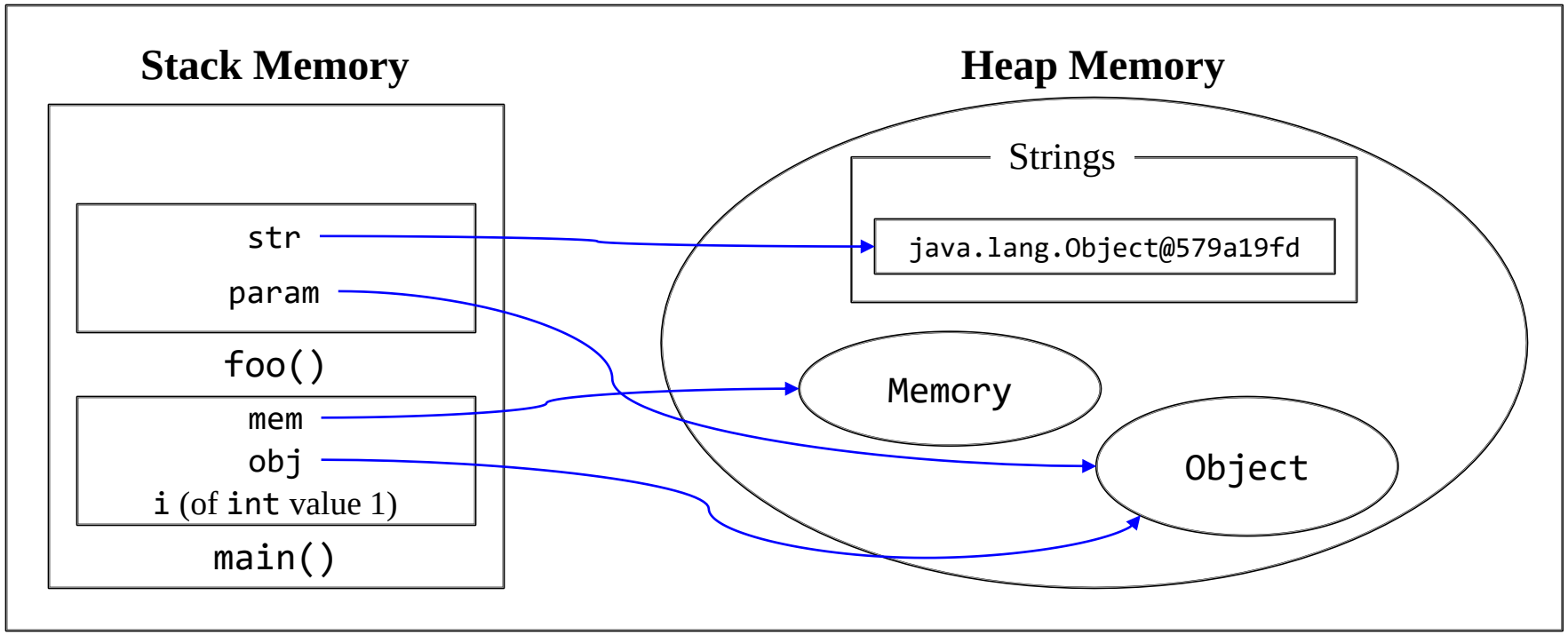
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

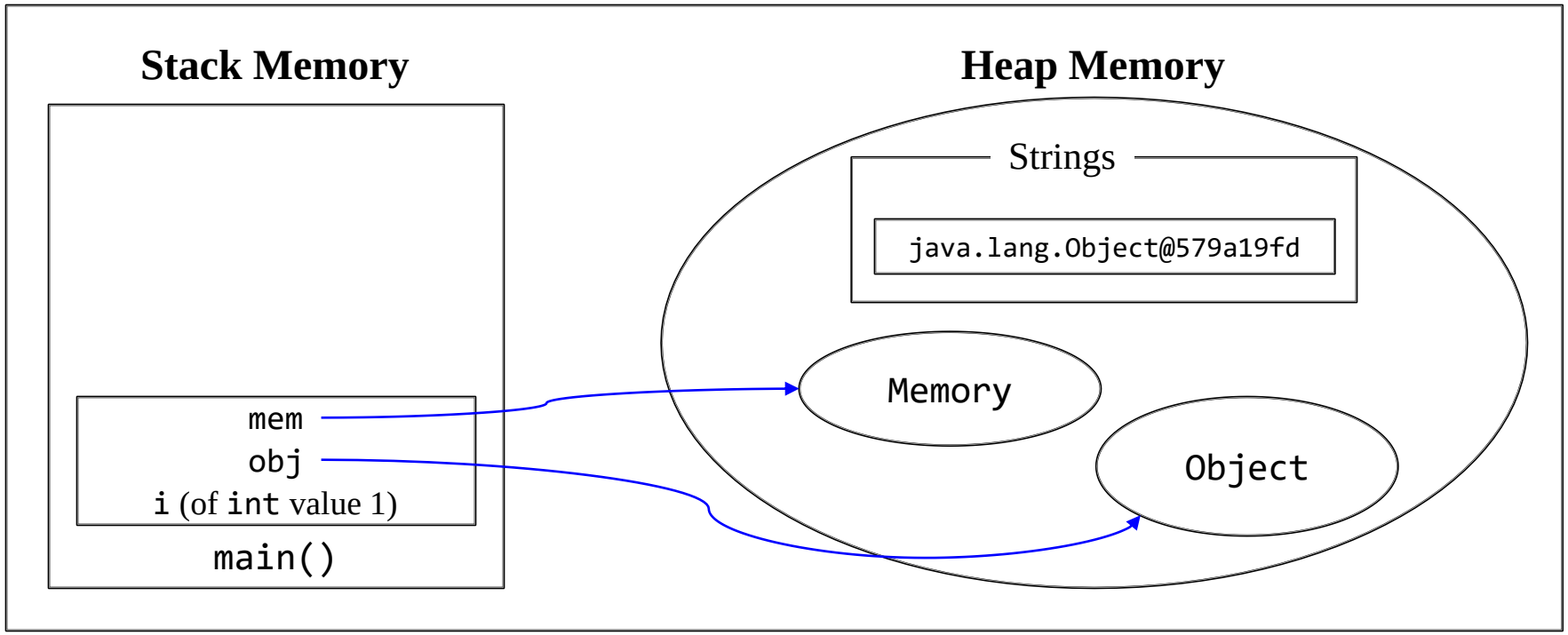
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

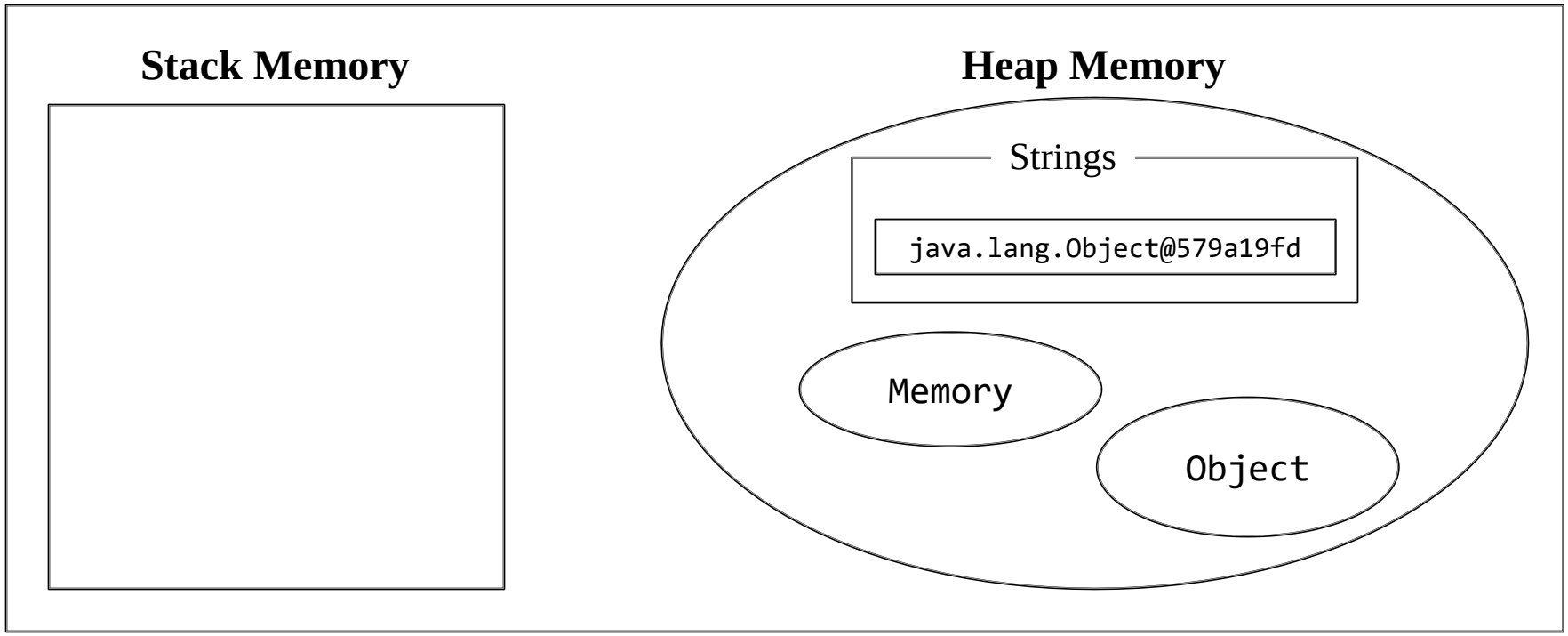
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

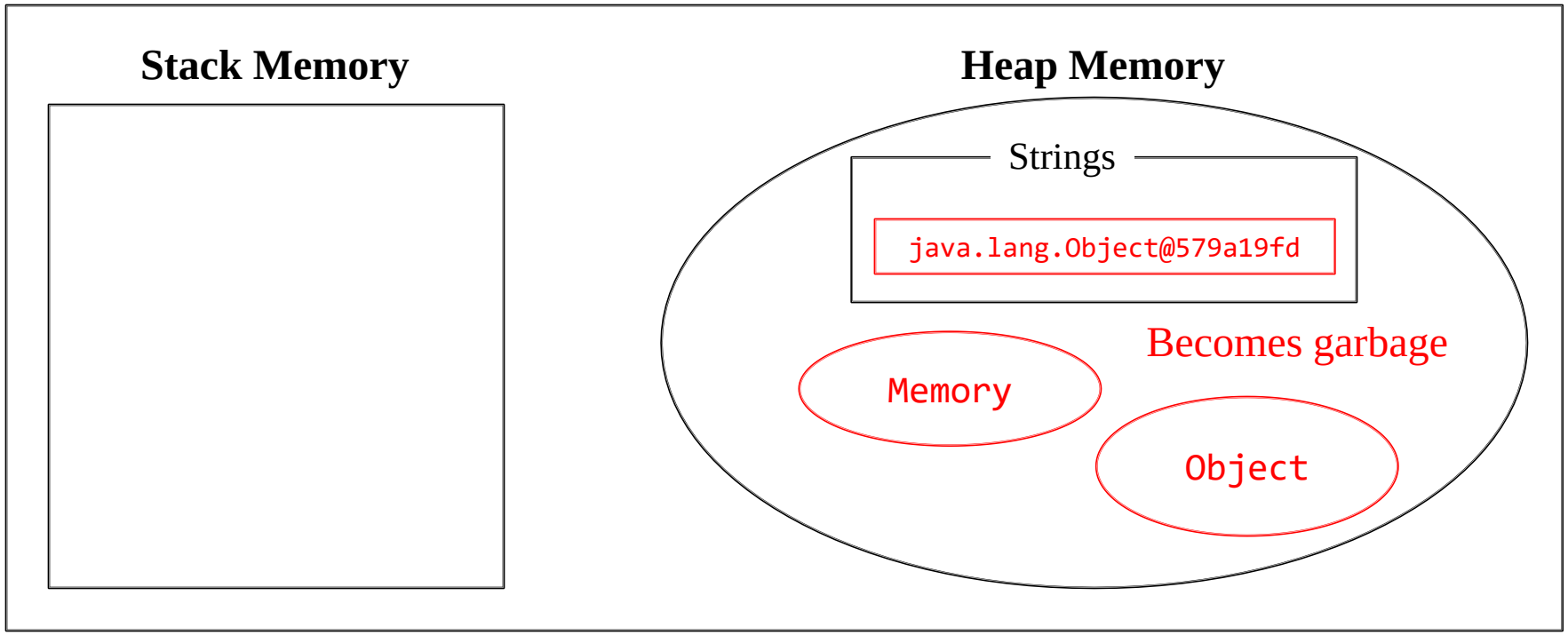
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

```
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory