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

Yepang LIU

liuyp1@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

- `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

`final` Instance Variables

- The principle of least privilege 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.
  - 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)

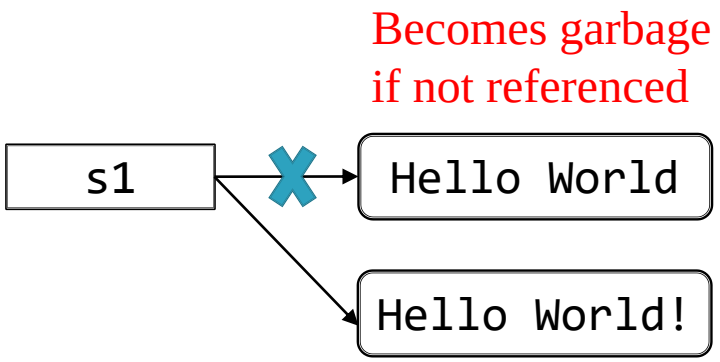
```
private final int INCREMENT;
```

- `final` variables can be initialized when they are declared.
- 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.

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).

Method `finalize`

- Every class has a method `finalize()`, which is inherited from the class `java.lang.Object`
- It is called by the garbage collector (GC) to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory
  - `finalize` does not take parameters and has return type `void`
  - A problem is that GC is not guaranteed to execute at a specified time (may never execute before a program terminates)
  - It's unclear if, or when, method `finalize` will be called
  - For this reason, most programmers should avoid method `finalize`

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.
- Packages facilitate software reuse by enabling programs to import classes from other packages, rather than copying the classes into each program that uses them.
- Packages provide a convention for unique class names, which helps prevent class-name conflicts.

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. Non-`public` classes are in a package to support the reusable classes in the package.

Creating Packages (Cont.)

- When a Java file containing a `package` declaration is compiled, the resulting class file is placed in the directory specified by the declaration.

- 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.
- 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

Specifying Classpath (Compilation)

- When compiling a class that uses classes from other packages, `javac` must locate the `.class` files for all these classes.
- The compiler uses a special object called a class loader to locate the classes it needs.
  - The class loader begins by searching the standard Java classes that are bundled with the JDK.
  - Then it searches for optional packages.
  - If the class is not found in the standard Java classes or in the extension classes, the class loader searches the classpath, which contains a list of locations in which classes are stored

Specifying Classpath (Compilation)

- 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
- 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 (Compilation)

- By default, the classpath consists only of the current directory
- The classpath can be modified by
  - providing the `-classpath (-cp)` option to the `javac` compiler
  - setting the `CLASSPATH` environment variable (not recommended).

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

Specifying Classpath (Execution)

- When you execute an application, the JVM must be able to locate the `.class` files of the classes used in that application.
- Like the compiler, the `java` command uses a class loader that searches the standard classes and extension classes first, then searches 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
```

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.

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)
  - Any attempt to create an object of an `enum` type with operator `new` results in a compilation error (constructor of an `enum` type can only be private or package-private, meaning without any access level modifier)
  - `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

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.

```
Direction d = Direction.NORTH;
System.out.println(d.toString()); // prints "NORTH"
```

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.

Example

```
import java.util.EnumSet;
    public class EnumTest {
    public static void main(String[] args) {
        System.out.println("All books:\n");

        for(Book book : Book.values())
            System.out.printf("%-10s%-45s%s\n", book,
                    book.getTitle(), book.getCopyrightYear());

        System.out.println("\nDisplay a range of enum constants:\n");

        for(Book book : EnumSet.range(Book.JHTP, Book.CPPHTP))
            System.out.printf("%-10s%-45s%s\n", book,
                    book.getTitle(), book.getCopyrightYear());
    }
}
```

`Values()` returns an array of the `enum`'s constants

`EnumSet`'s method `range()` returns a collection of the `enum` constants in the specified range of constants

Example

```
All books:

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

Display a range of enum constants:

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
```

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

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).

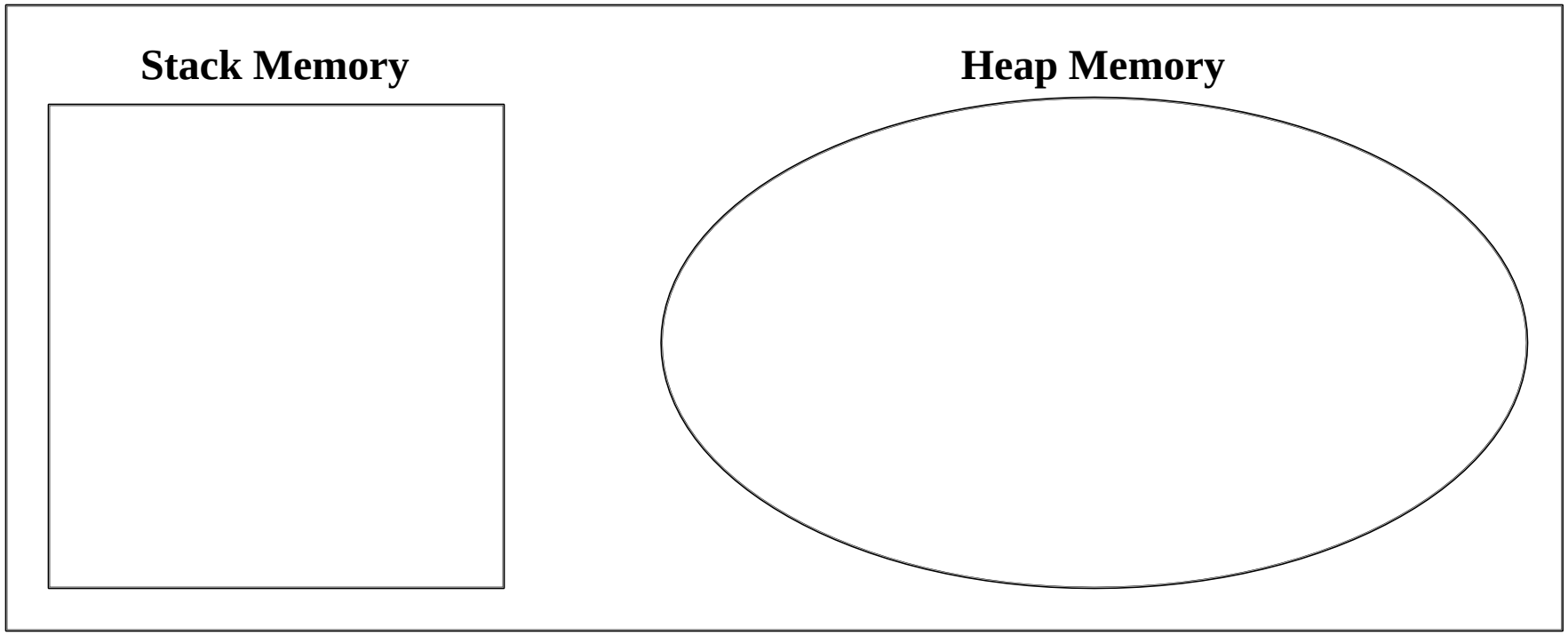
https://www.journaldev.com/4098/java-heap-space-vs-stack-memory

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);
}
```

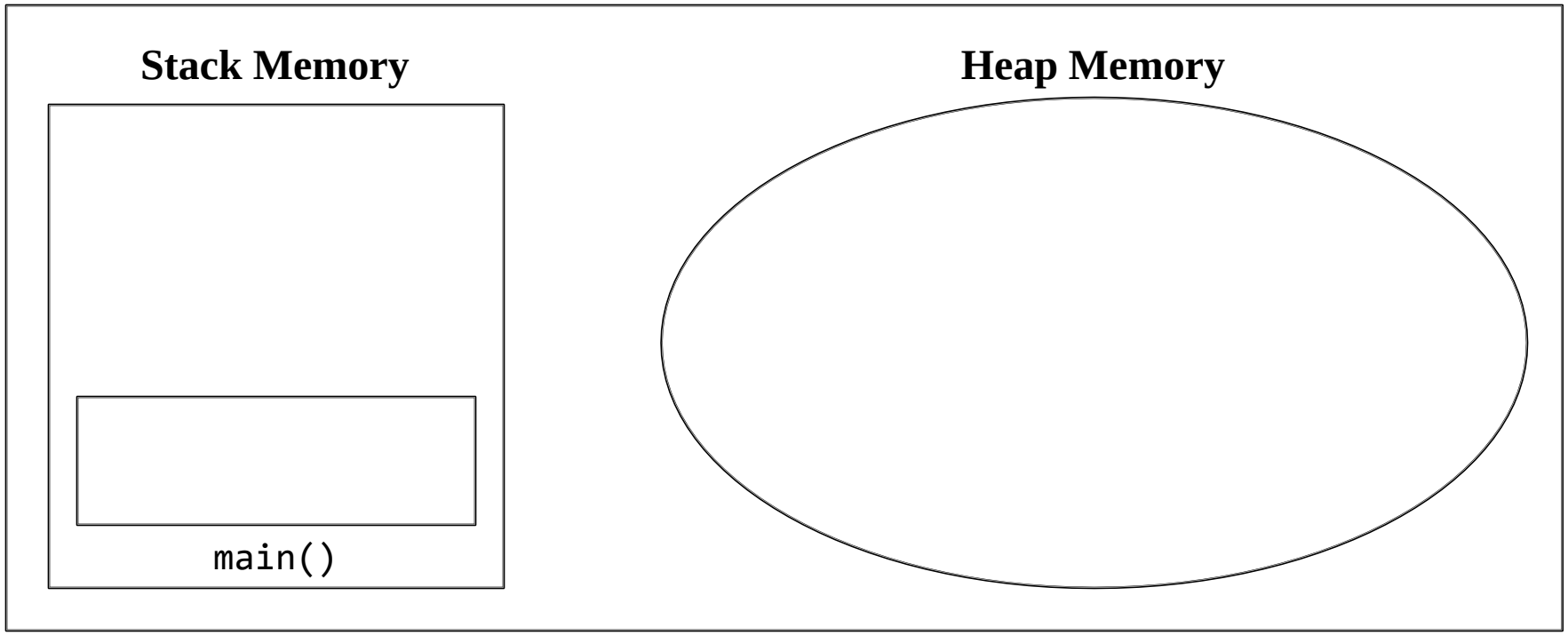
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

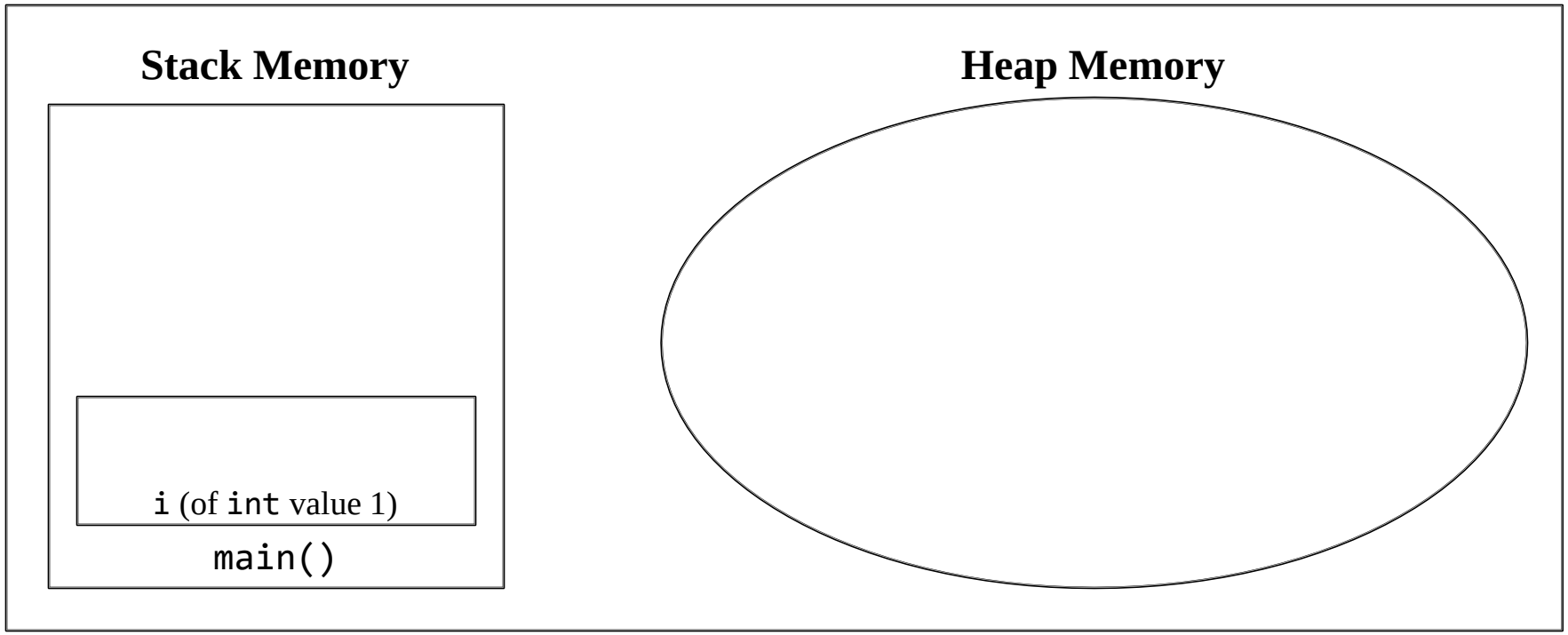
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

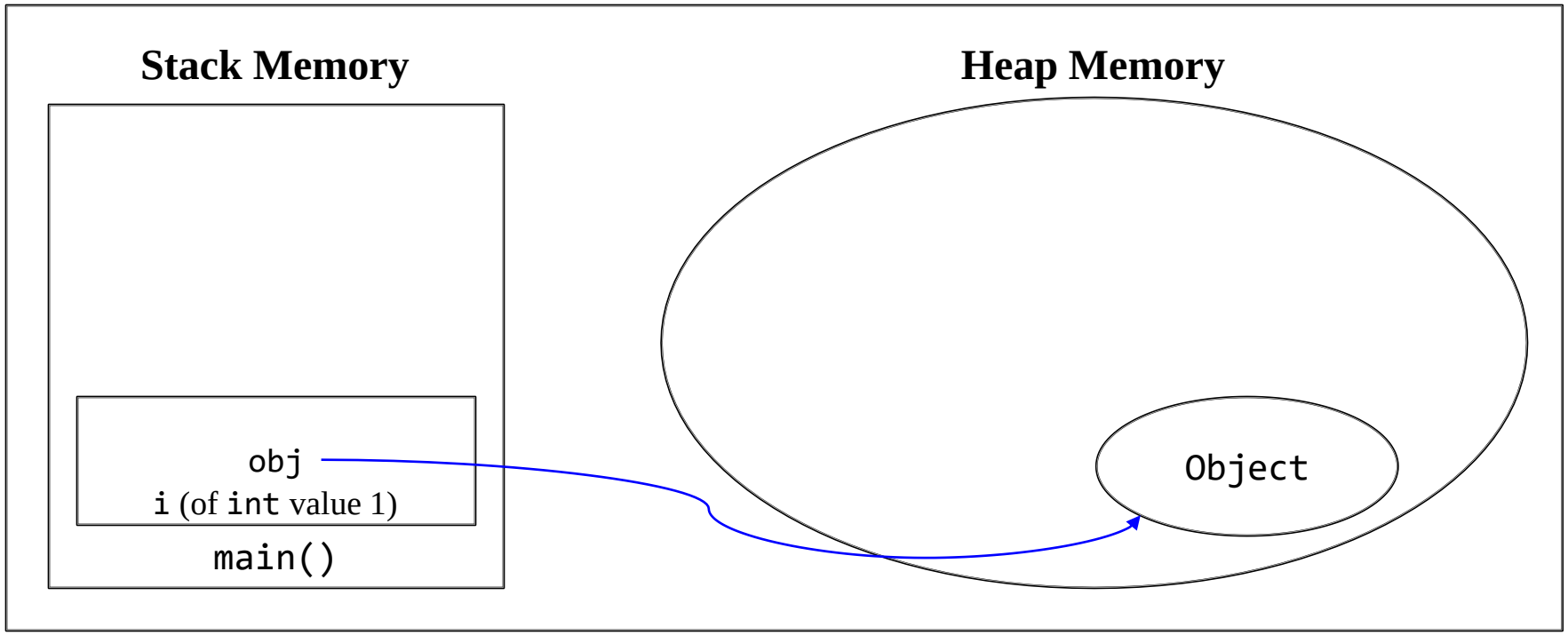
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

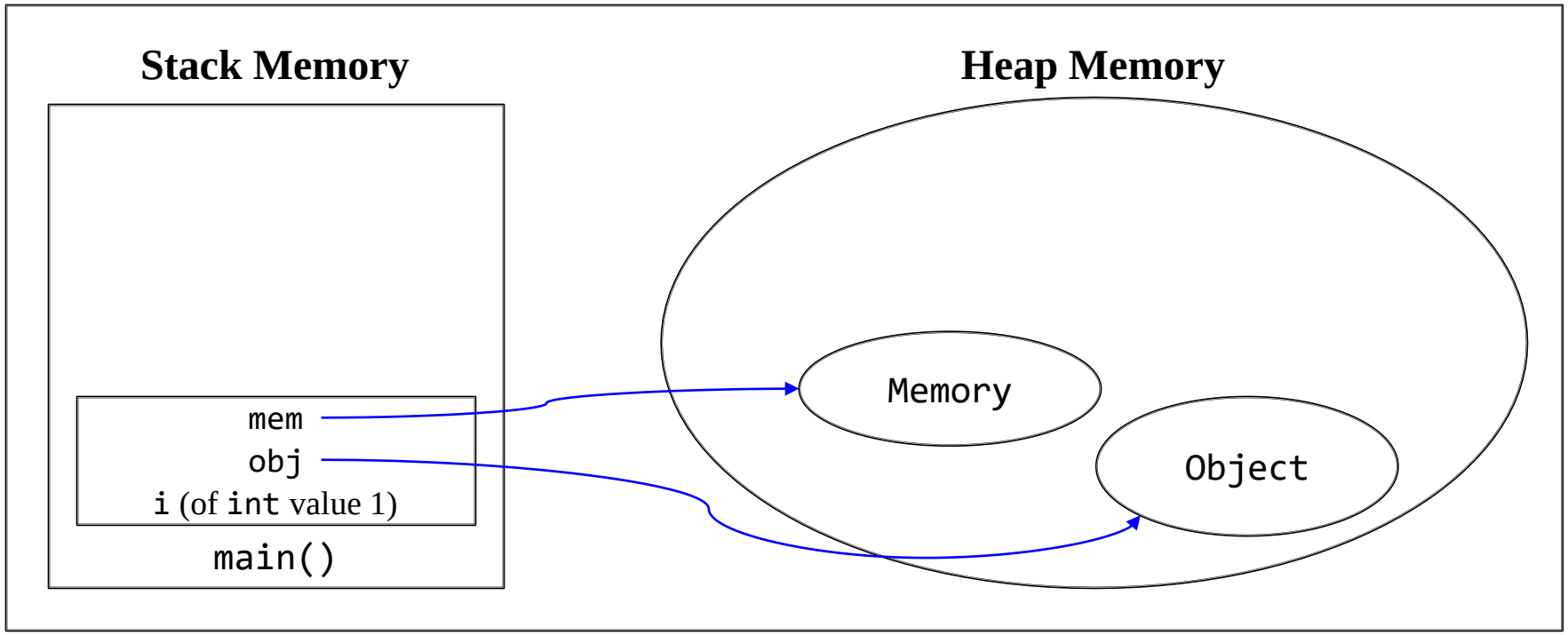
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

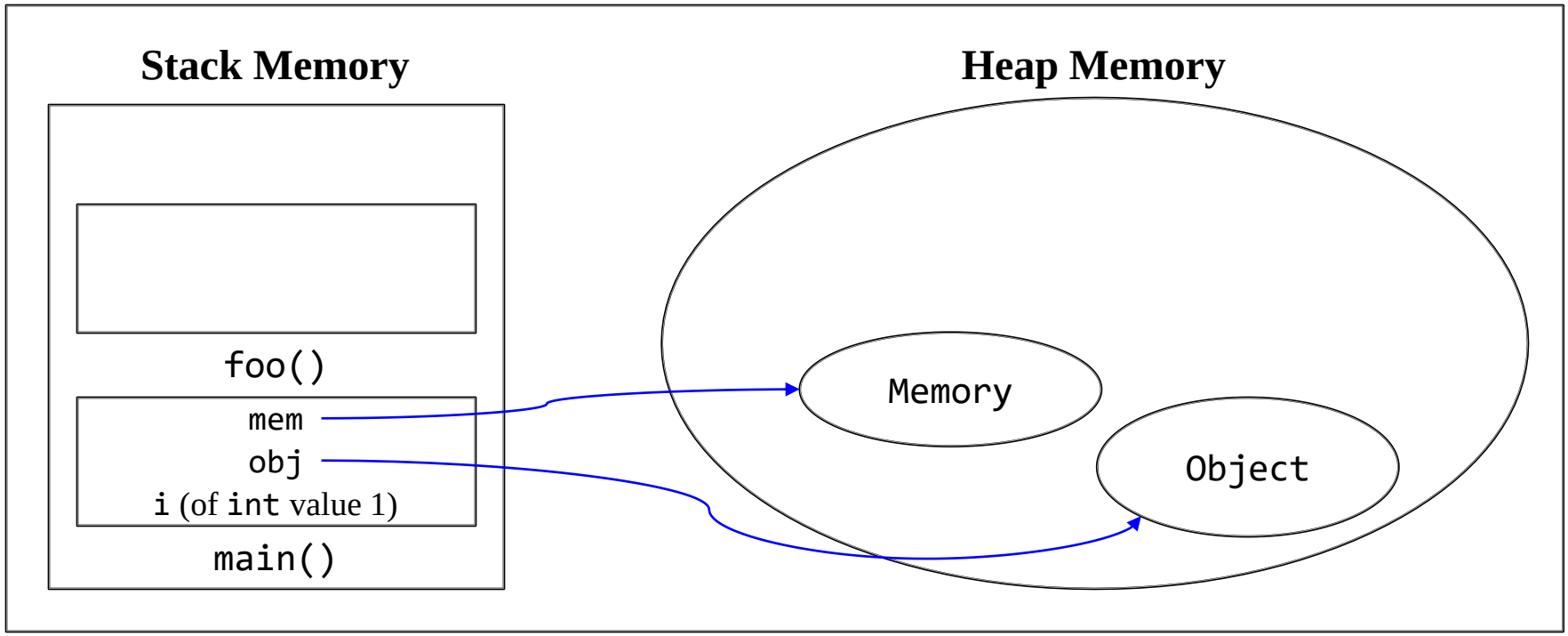
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

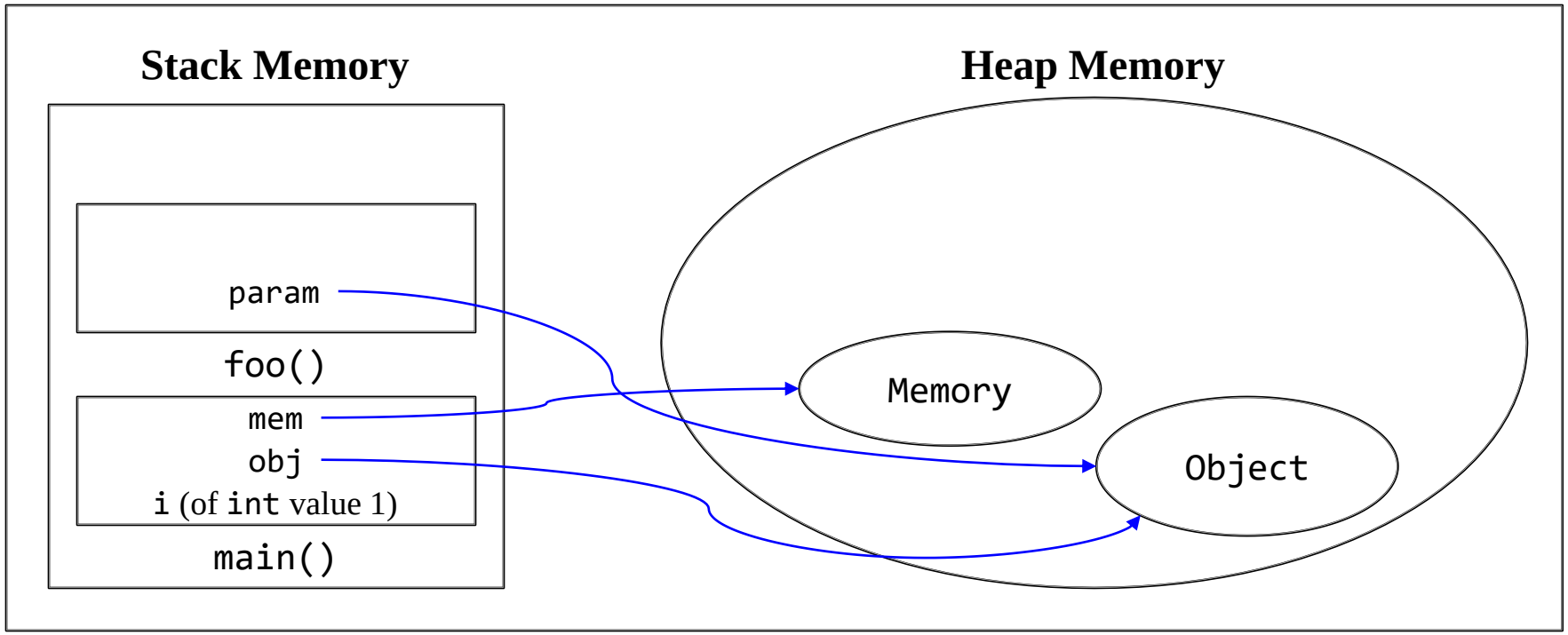
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

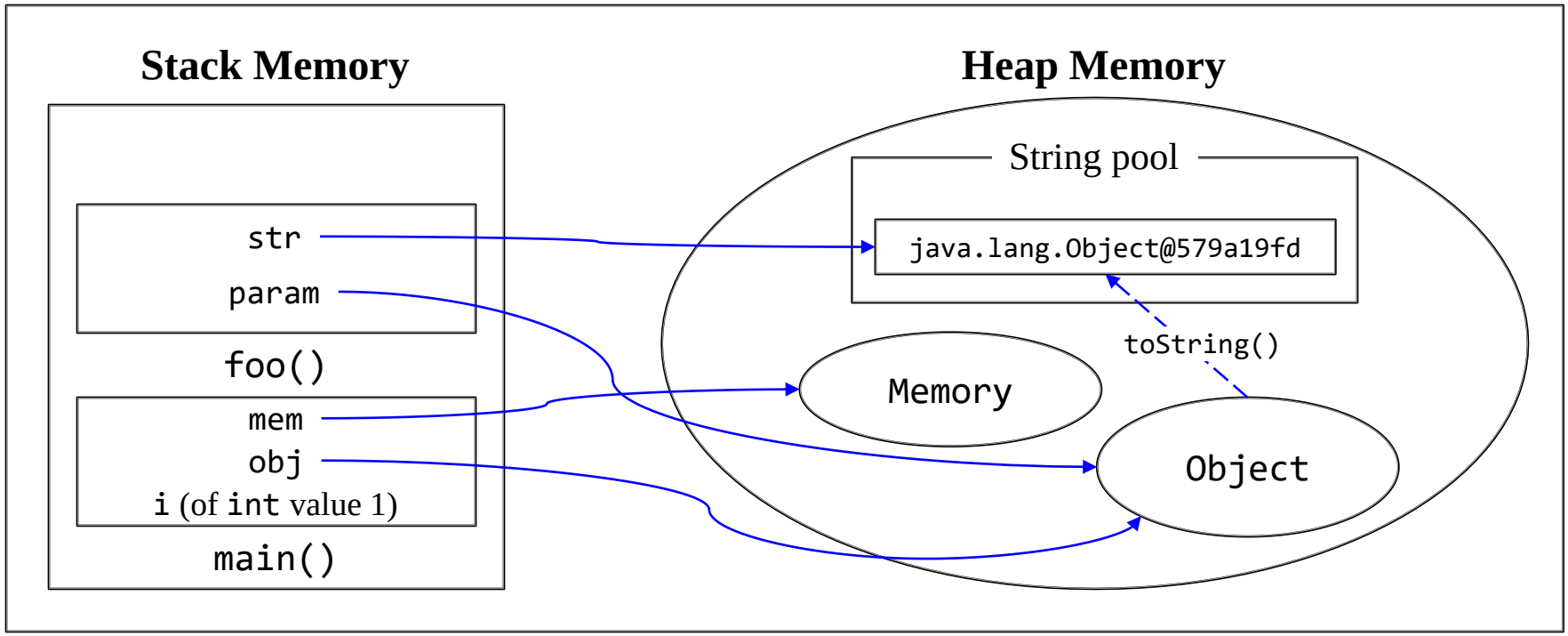
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

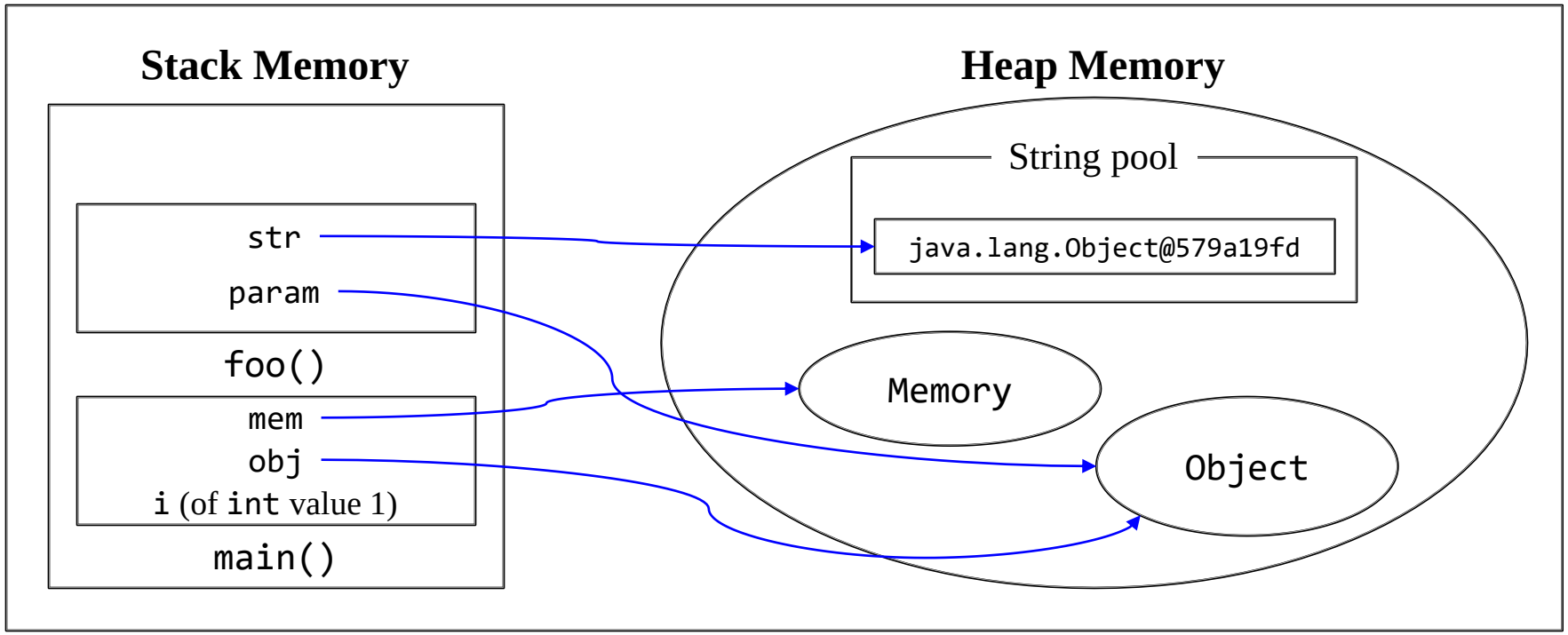
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

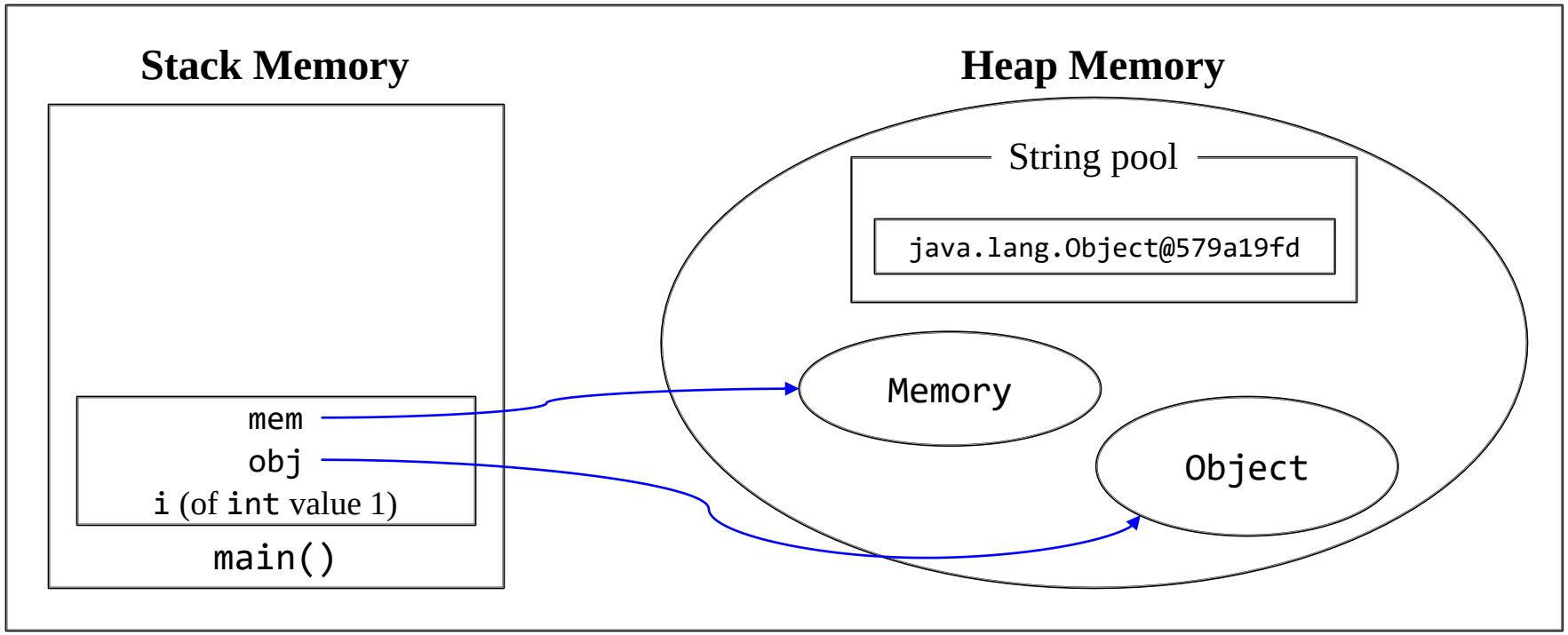
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

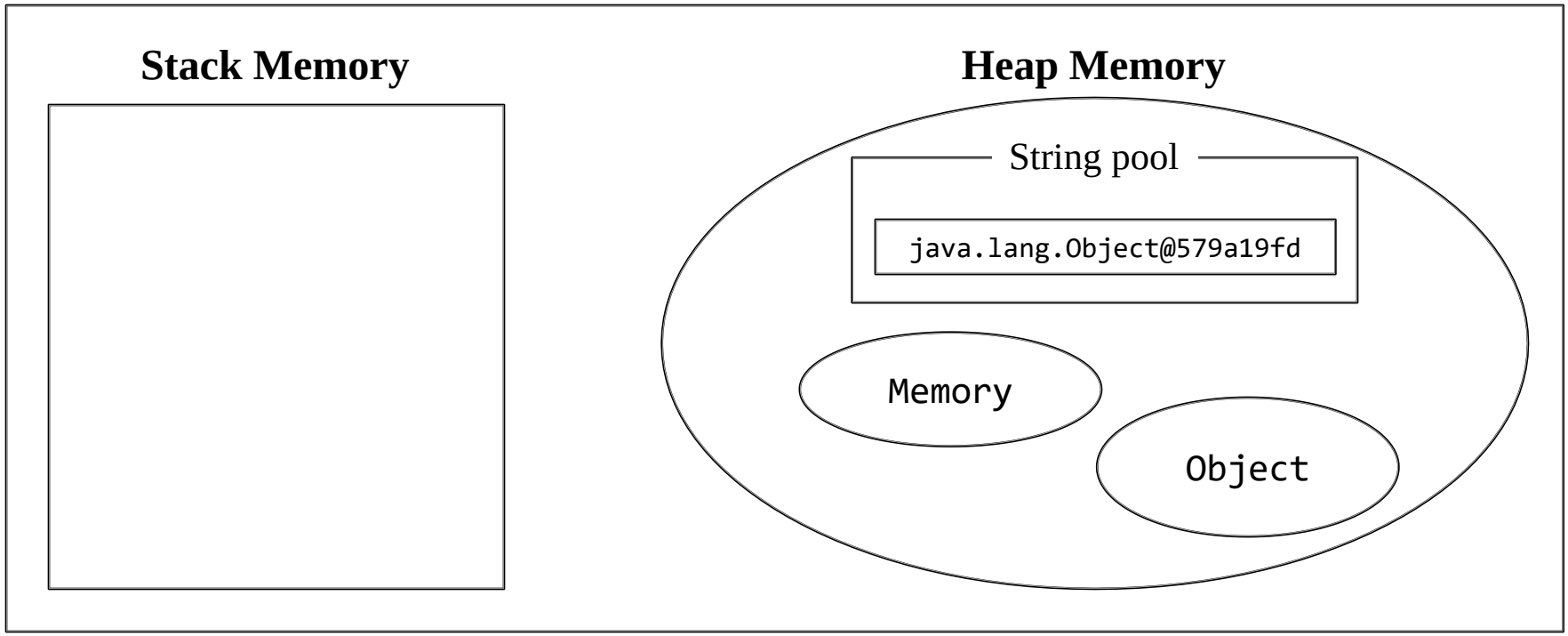
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

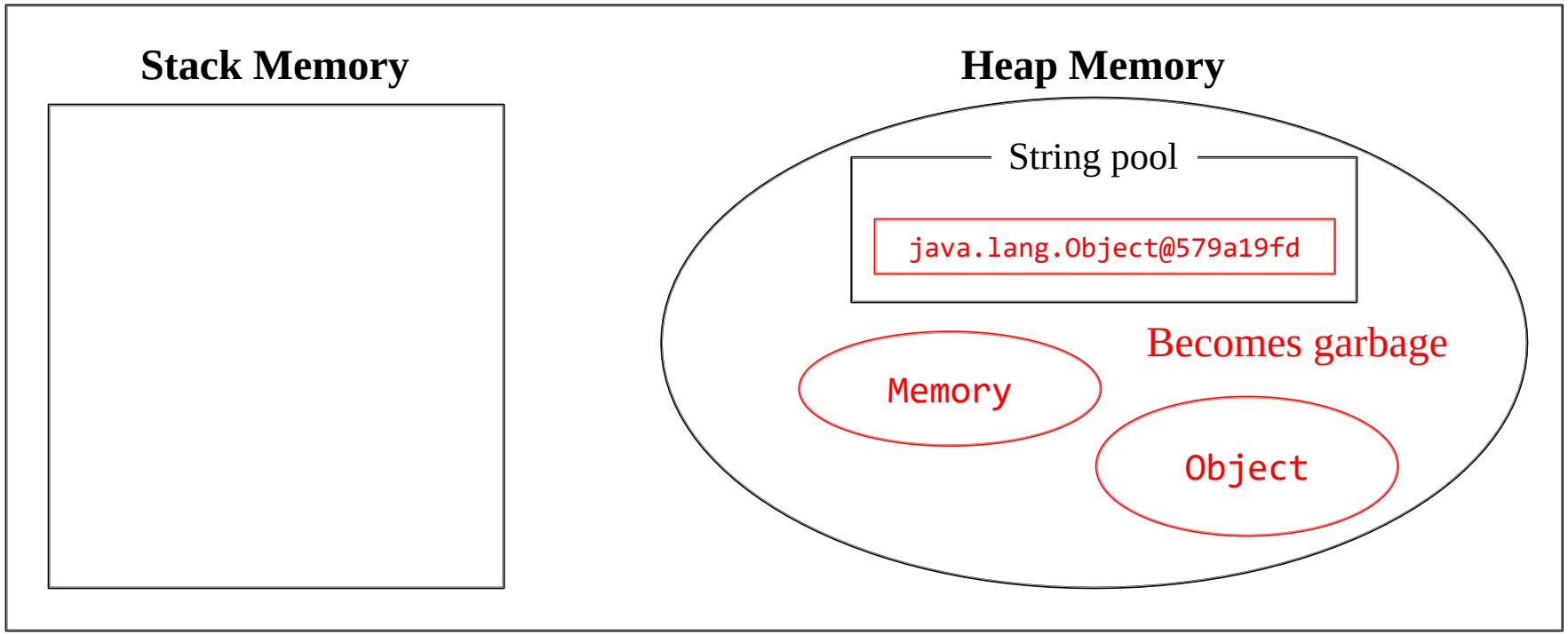
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



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