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

TAO Yida

taoyd@sustech.edu.cn

Objectives

- Packages & Package access
- The `final` keyword & Enumerations
- Wrapper classes, ArrayList
- Stack and heap memory

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.cs101;

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

Declare package

- A Java package structure is like a directory structure. Its a tree of packages, subpackages and classes inside these classes.

- The class `Time` should be placed in the directory. It can be identified by its fully qualified name: `sustech.cs101.Time`

```
sustech
    cs101
        Time.java
```

```
package sustech.cs101;

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

Importing a class

- A single-type-import declaration specifies one class to import, then we can use its simple name (e.g., `Time`) directly in the program.

- A java file must have the following order:
  - A `package` declaration (if any)
  - `import` declarations (if any)
  - class declarations

```
package lecture9;

import sustech.cs101.Time;
import java.util.Scanner;

public class Client {
    Time time;
    Scanner sc;
}
```

Importing a class

- If we don't import, we have to use the fully qualified name in programs
- Reason: 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

```
package lecture9;

//import sustech.cs101.Time;
//import java.util.Scanner;

public class Client {
    sustech.cs101.Time  time;
    java.util.Scanner  sc;

}
```

Importing a class

- If we don't import, we have to use the fully qualified name in programs
- Reason: 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

- String belongs to the `java.lang` package.
- `java.lang` mostly includes essential features and all classes in this package are imported implicitly by default (we don't need to do the import by ourselves)

```
package lecture9;

import sustech.cs101.Time;
import java.util.Scanner;

public class Client {
    Time time;
    Scanner sc;
    String s;


}
```

Importing multiple classes

- 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 all classes in java.util`

- 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 ImportDemo {
    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.

Package Access for Class Members

- 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

Controlling access to class members (fields & methods).

Example

```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

Access Modifiers for Classes

- At the top level, a class can only be declared as `public` or package-private (no explicit modifier).
- A top-level class cannot be private.

- We 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 must not have public access modifiers, and can be used only by the other classes in the package (package-private).

Example

```
package p1;

class C1 {
   ...
}
```

```
package p1;

public class C2 {
   can access C1
}
```

```
package p2;

public class C3 {
   cannot access C1;
   can access C2;
}
```

Objectives

- Packages & Package access
- The `final` keyword & Enumerations
- Wrapper classes, ArrayList
- Stack and heap memory

The `final` keyword

- In Java, the `final` keyword is used to denote constants
- The `final` keyword can be used with variables, methods (later), and classes (later).
- Once any entity (variable, method or class) is declared `final`, it can be assigned only once, i.e., it cannot be altered after initialization

`final` (local) variables

```
public class FinalDemo {
    public static void main(String[] args) {
        final int AGE = 32;
        AGE = 45;
    }
}
```

Cannot assign a value to final variable 'AGE'

Make 'AGE' not final Alt+Shift+Enter More a

`final` fields

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

```
private final int I_AM_A_CONSTANT = 2;
```

- Generally, every field in an object or class is initialized to a zero-like value during the allocation of memory.

- However, there's 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` fields

- Any final field must be initialized before the constructor completes.
  - `final` variables can be initialized when they are declared.

    ```
    public static final double PI = 3.14159265358979323846;
    ```

  - 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.

Enumerations (枚举)

- 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
- The identifiers in an `enum` must be unique
- Because they are constants, the names of an enum type's fields are in uppercase letters.

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

Enumerations

- Like classes, all `enum` types are reference types
- Once a enum type is defined, you can declare a variable of that type.
- Variables of the type `Direction` can be assigned only the four constants declared in the enumeration or null (other values are illegal, won't compile)

```
Direction d = Direction.EAST;
```

Enumerations

- The last statement is equivalent to `System.out.println(d.toString())`.

- When an enum constant is converted to a String using `toString()`, the constant's identifier is used as the String representation.

```
Direction d = Direction.EAST;

System.out.println(d);  // print "EAST"
```

Enumerations

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

```
public class DirectionTest {

    private Direction direction;

    public DirectionTest(Direction direction) {
        this.direction = direction;
    }

    public void tellDirection() {
        switch(direction) {
            // must be unqualified name of the enum constant
            case EAST:
                System.out.println("East direction");
                break;
            case WEST:
                System.out.println("West direction");
                break;
            case SOUTH:
                System.out.println("South direction");
                break;
            case NORTH:
                System.out.println("North direction");
                break;
        }
    }

    public static void main(String[] args) {
        DirectionTest myDirection = new DirectionTest(Direction.EAST);
        myDirection.tellDirection();
    }
}
```

Enumerations under the hood

- Each `enum` declaration declares an `enum` class with the following restrictions:
  - An enum constructor cannot be public; Any attempt to create an object of an `enum` type with operator `new` results in a compilation error (no other Direction instances can be created at runtime)
  - EAST and WEST are enum constants, which are instances of the Direction type
  - `enum` constants are implicitly `static` (no instance is needed to access them)
  - `enum` constants are implicitly `final` (constants that shouldn't be modified)

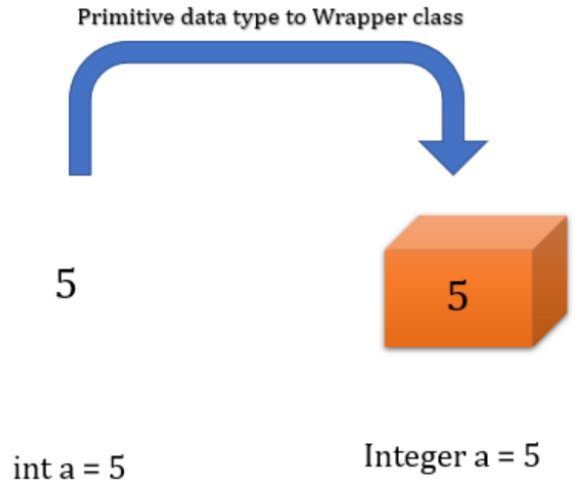
```
public final class Direction {
   private Direction() {}          // private constructor so that no one else could create new Direction objects
   public static final Direction EAST = new Direction();
   public static final Direction WEST = new Direction();
   public static final Direction SOUTH = new Direction();
   public static final Direction NORTH = new Direction();
}
```

Objectives

- Packages & Package access
- The `final` keyword & Enumerations
- Wrapper classes, ArrayList
- Stack and heap memory

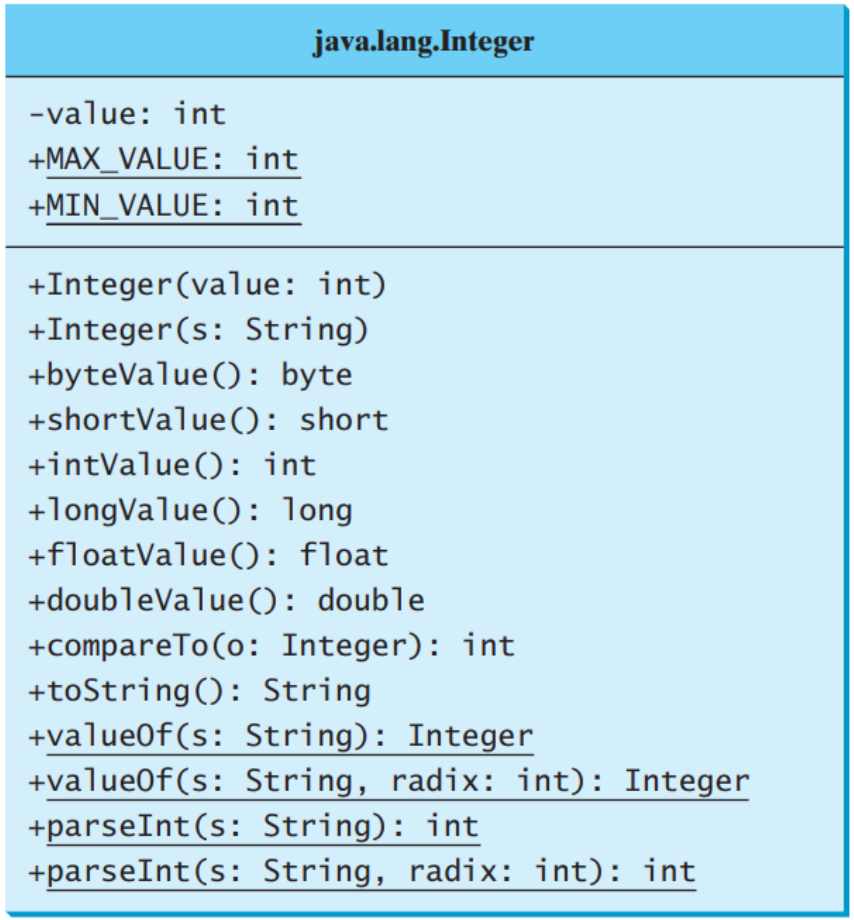
Wrapper Classes

- Java has 8 primitive types: `boolean`, `char`, `double`, `float`, `byte`, `short`, `int` and `long`
- Java also provides 8 type-wrapper classes—`Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` and `Long`—that enable primitive-type values to be treated as objects.



Using Wrapper Classes

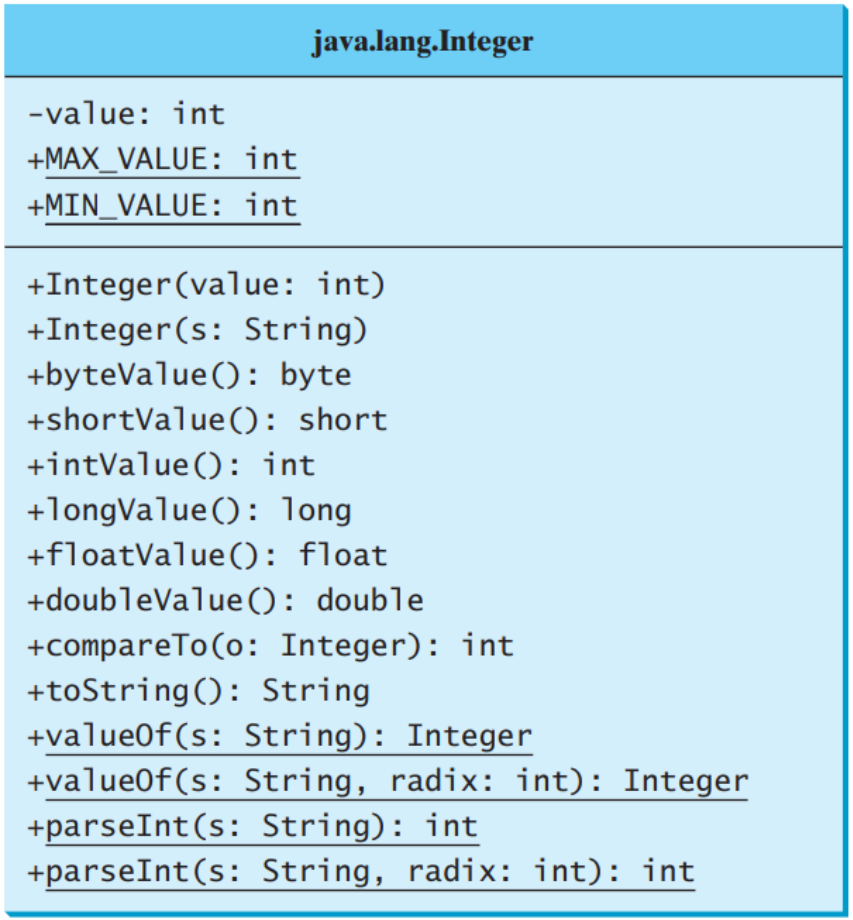
```
Integer x1 = Integer.valueOf("32");
Integer x2 = 32;

int i = Integer.parseInt("123");
double d = Double.parseDouble("12.345");
```



Using Wrapper Classes

- Each numeric wrapper class has the constants MAX_VALUE and MIN_VALUE
- Each numeric wrapper class contains the methods doubleValue(), floatValue(), intValue(), longValue(), and shortValue() for returning a double, float, int, long, or short value for the wrapper object

```
Double.valueOf(12.4).intValue(); // 12
Integer.valueOf(12).doubleValue(); // 12.0
```



Useful Character Methods

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter a character and press Enter:");
String input = sc.next();
char c = input.charAt(0);

System.out.printf("is digit: %b\n", Character.isDigit(c));
System.out.printf("is identifier start: %b\n", Character.isJavaIdentifierStart(c));
System.out.printf("is letter: %b\n", Character.isLetter(c));
System.out.printf("is lower case: %b\n:", Character.isLowerCase(c));
System.out.printf("is upper case: %b\n", Character.isUpperCase(c));
System.out.printf("to upper case: %c\n", Character.toUpperCase(c));
System.out.printf("to lower case: %c\n", Character.toLowerCase(c));

sc.close();
```

Useful Character Methods

```
Enter a character and press Enter:
A
is digit: false
is identifier start: true
is letter: true
is lower case: false
is upper case: true
to upper case: A
to lower case: a
```

```
Enter a character and press Enter:
8
is digit: true
is identifier start: false
is letter: false
is lower case: false
is upper case: false
to upper case: 8
to lower case: 8
```

Java identifiers can only star with a letter, an underscore (_), or a dollar sign ($)

Boxing & Unboxing

- Converting a primitive value to a wrapper object is called boxing.

  For example, converting an `int` to an `Integer`, a `double` to a `Double`, and so on.

- The reverse conversion is called unboxing.

  For example, converting an `Integer` to an `int`, a `Double` to a `double`, and so on.

Autoboxing & Auto-unboxing

The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value. This is called autoboxing and autounboxing.

```
Integer[] intArray = {1, 2, 3};
```

The primitive values 1,2, and 3 are automatically boxed into objects new Integer(1), new Integer(2), and new Integer(3).

```
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

The objects intArray[0], intArray[1], and intArray[2] are automatically unboxed into int values that are added together

ArrayList

- Arrays store sequences of objects (and primitive values). Arrays do not change their size at runtime to accommodate additional elements.
- `ArrayList<T>` can dynamically change its size at runtime.
- `ArrayList<T>` is a **generic class**, where `T` is a placeholder for the type of elements that you want the `ArrayList` to hold.

```
ArrayList<String> list;
```

Declares `list` as an `ArrayList` collection to store only `String` objects

ArrayList

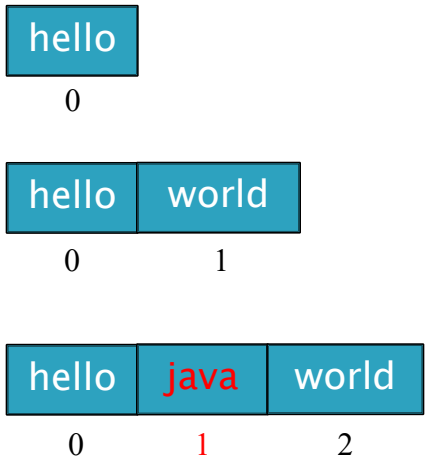
- Arrays store sequences of objects (and primitive values). Arrays do not change their size at runtime to accommodate additional elements.
- `ArrayList<T>` can dynamically change its size at runtime.
- `ArrayList<T>` is a **generic class**, where `T` is a placeholder for the type of elements that you want the `ArrayList` to hold.

```
ArrayList<int> list;
```

`ArrayList` cannot hold primitive data types

```
public class ArrayListDemo {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();

        // add element to the end of the list
        list.add("hello");
        printList(list);

        list.add("world");
        // get element at specified index
        String s = list.get(1);
        System.out.println(s);

        // add element at specified index
        list.add(1, "java");
        System.out.println(list.get(1));
        printList(list);
    }
    public static void printList(ArrayList<String> list) {
        // traverse the list using enhanced for loop
        for(String s : list) {
            System.out.printf("%s ", s);
        }
        System.out.println();
    }
}
```

Using ArrayList



```
public class SortDemo {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(5); // autoboxing
        list.add(124);
        list.add(-8);
        printList(list);

        // sort elements in the list
        // by ascending order
        Collections.sort(list);

        printList(list);
    }
    public static void printList(ArrayList<Integer> list) {
        for(Integer i : list) {
            // autounboxing
            System.out.printf("%d ", i);
        }
        System.out.println();
    }
}
```

Sort ArrayList

`java.util.Collections` class provides ***static*** methods that operate on list (e.g., `shuffle, reverse, sort`)

Objectives

- Wrapper classes
- Packages
- Class member access levels
- The `final` keyword
- Enumerations
- ArrayList
- Stack and heap memory

Java Heap Memory

- The heap space is used by Java runtime to allocate memory to Objects. 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)

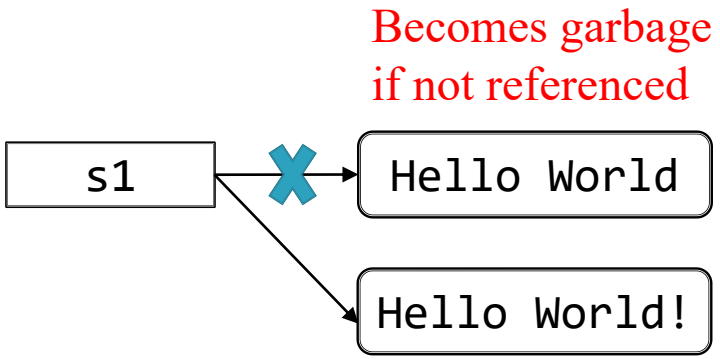
- Garbage Collection runs on the heap memory to free the memory used by objects that doesn't have any reference.

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("!");
```



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).

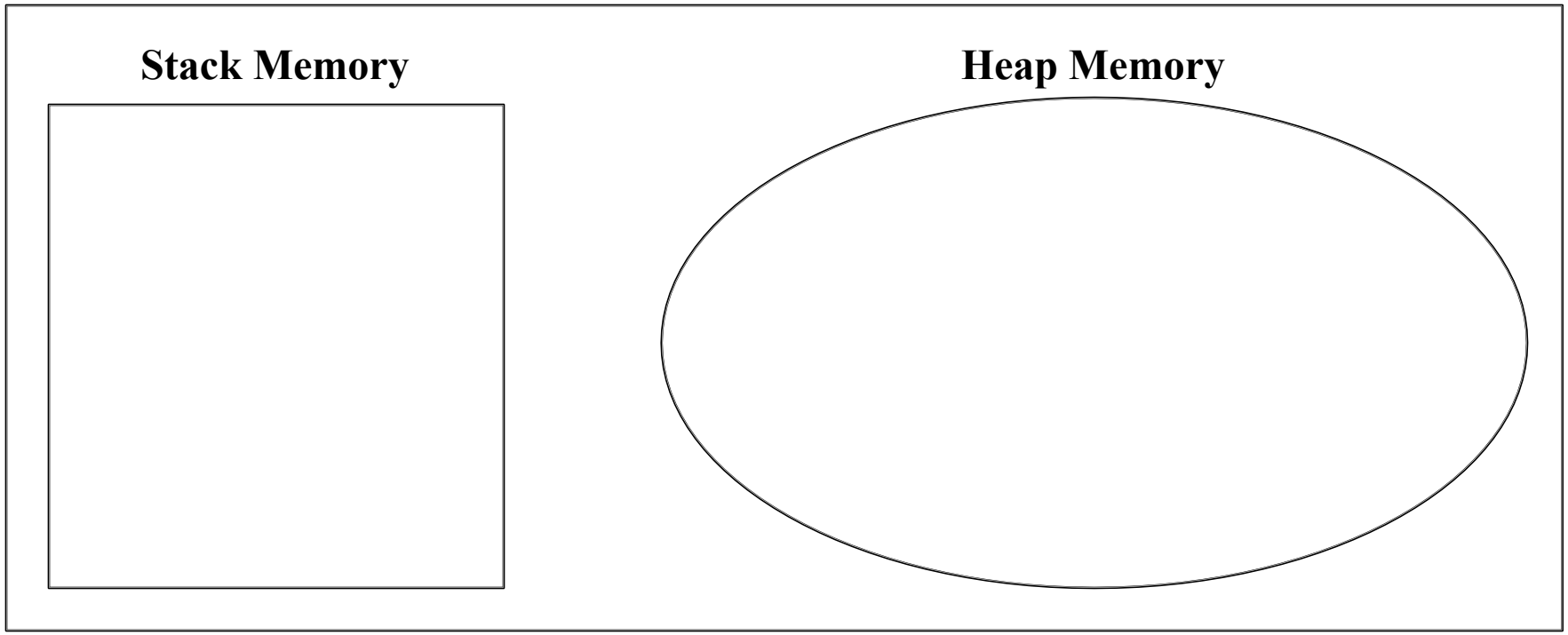
https://www.journaldev.com/4098/java-heap-space-vs-stack-memory

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);
}
```

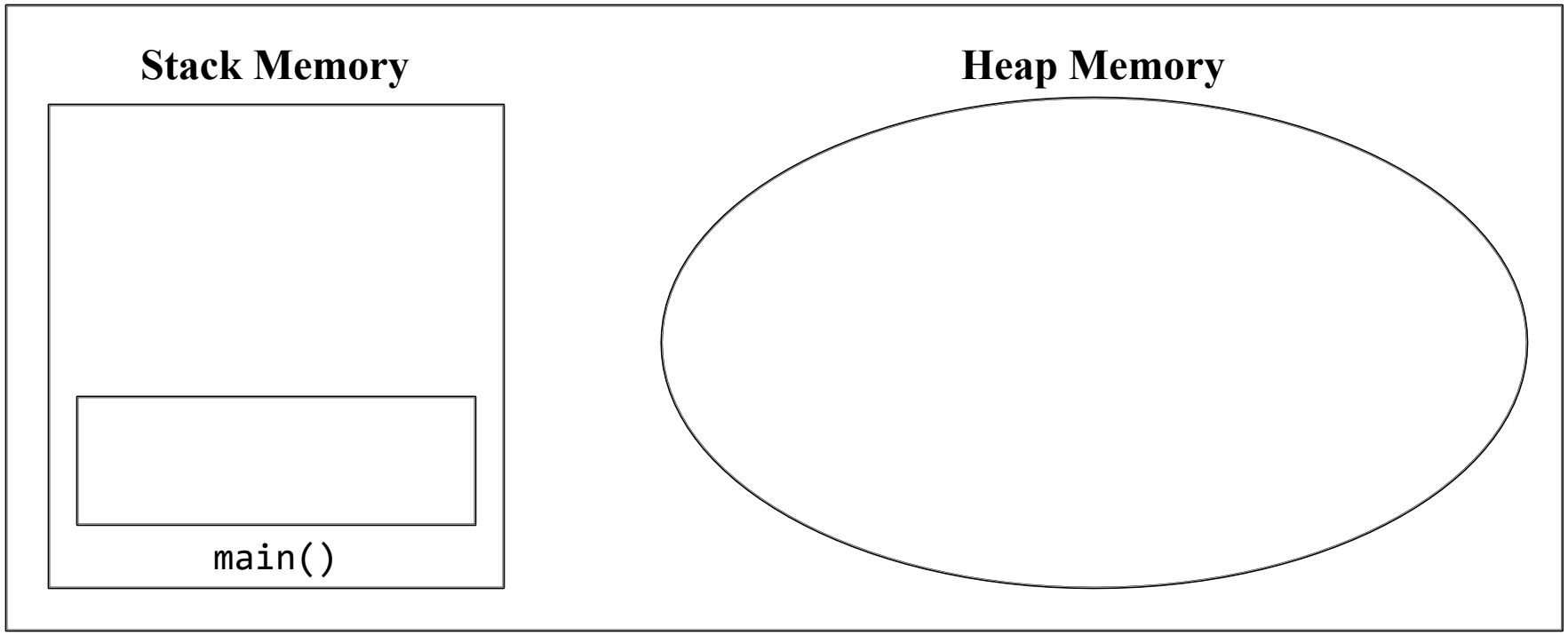
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

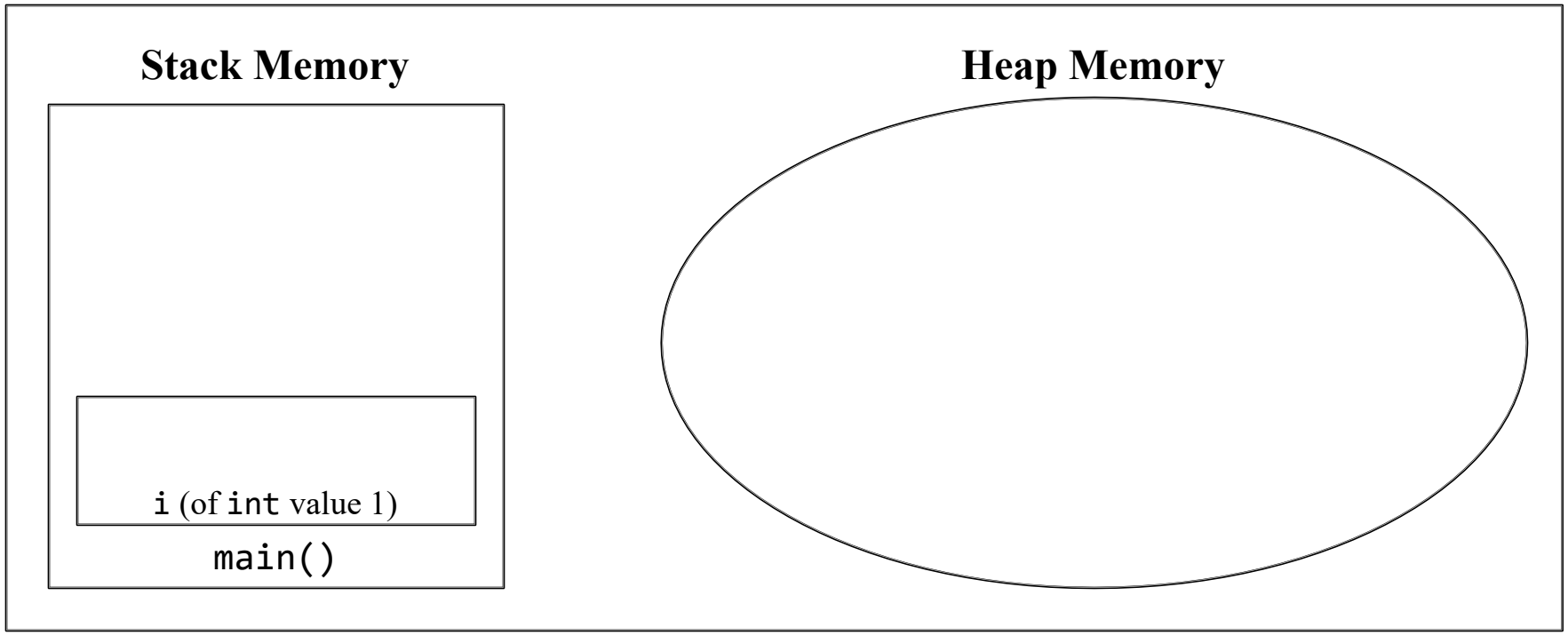
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

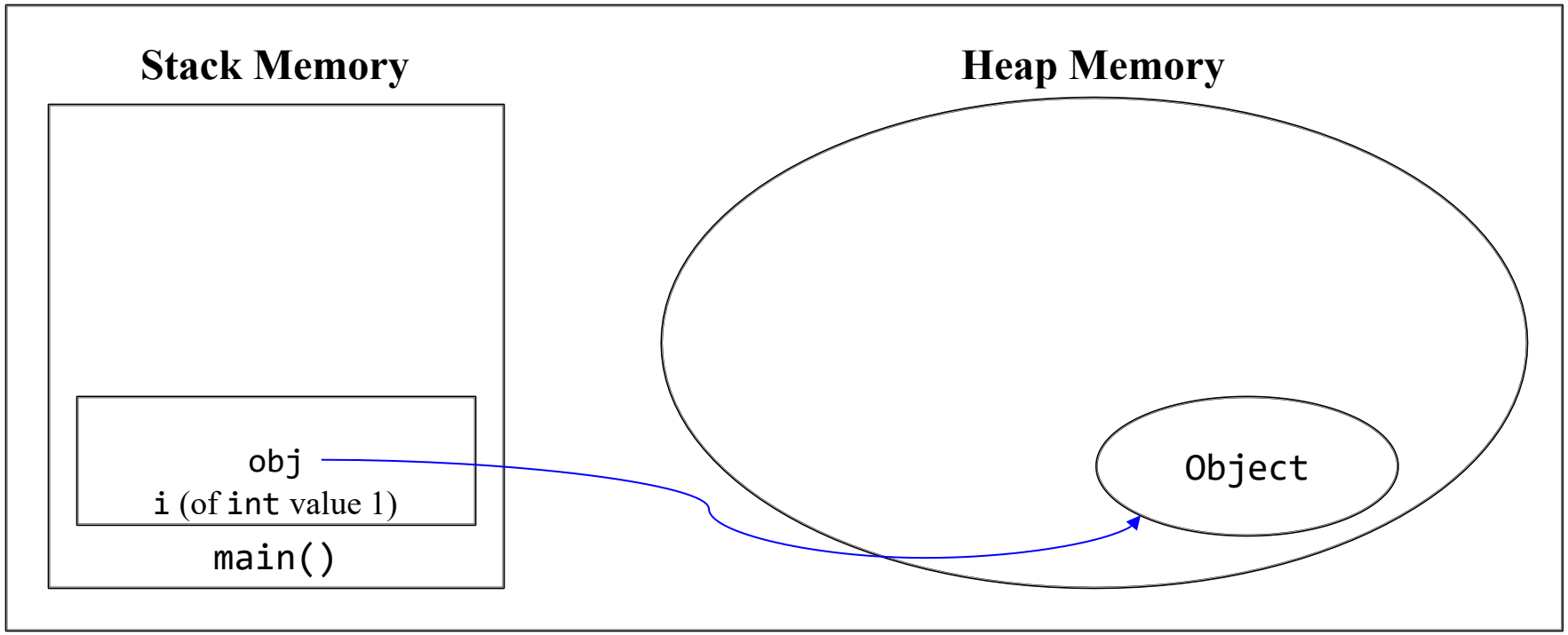
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

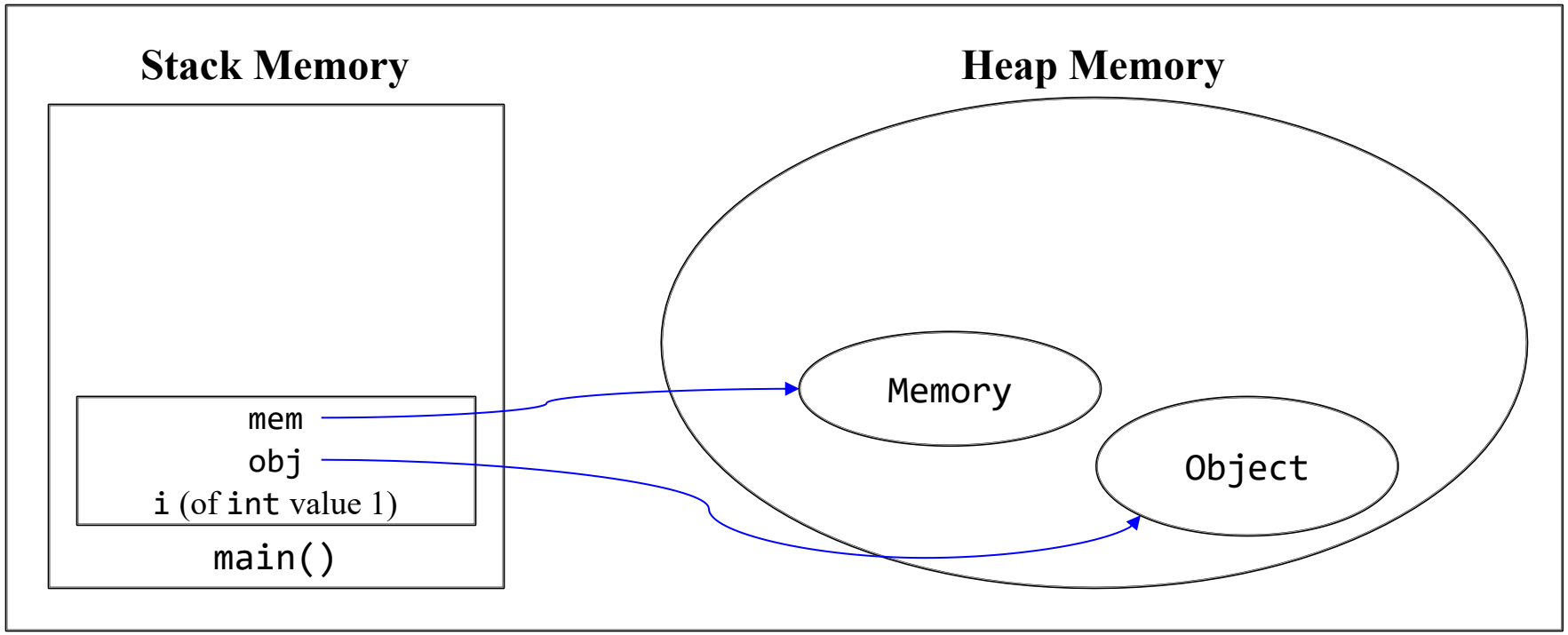
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

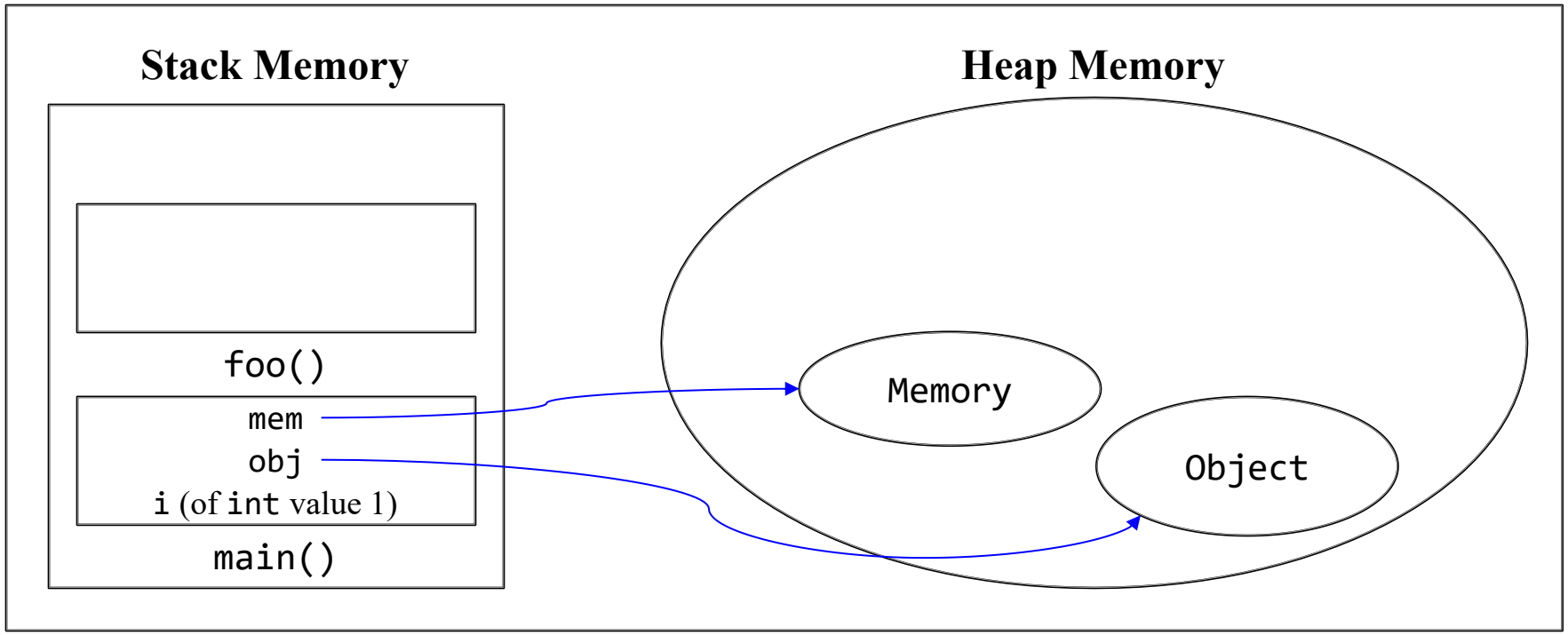
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

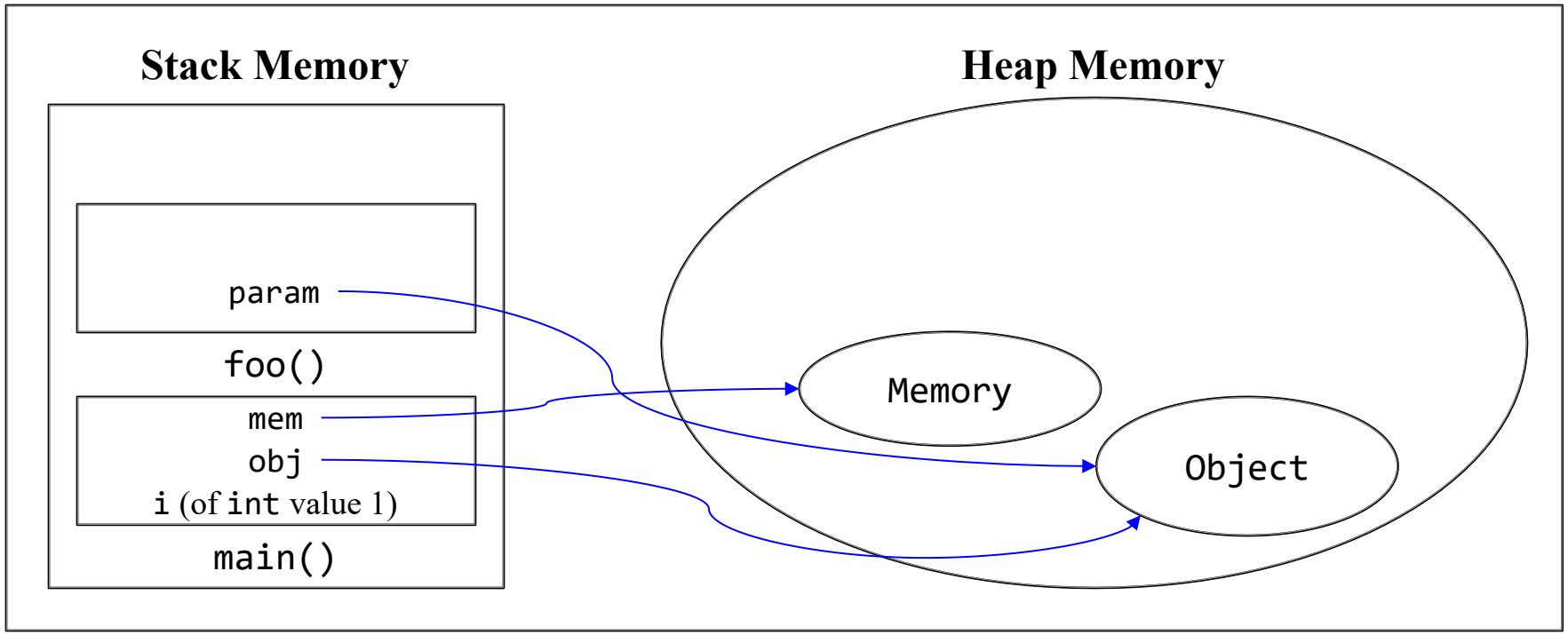
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

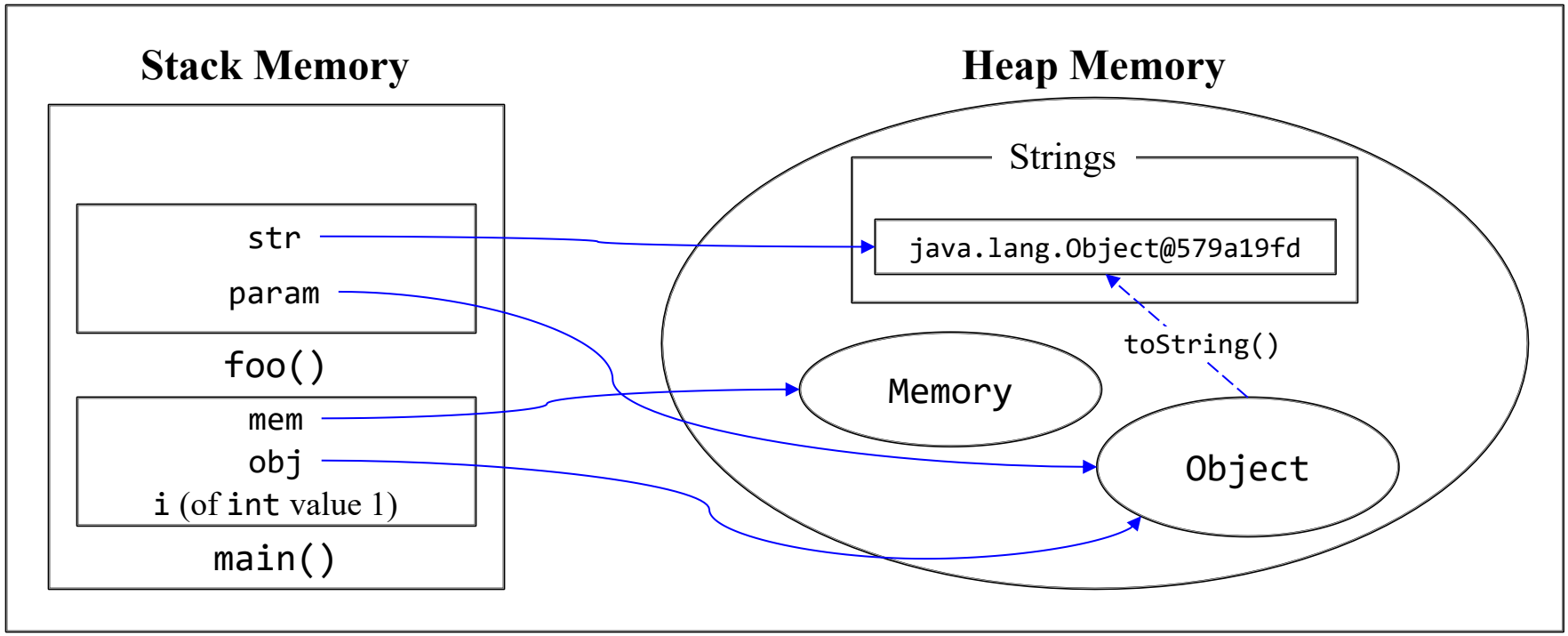
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

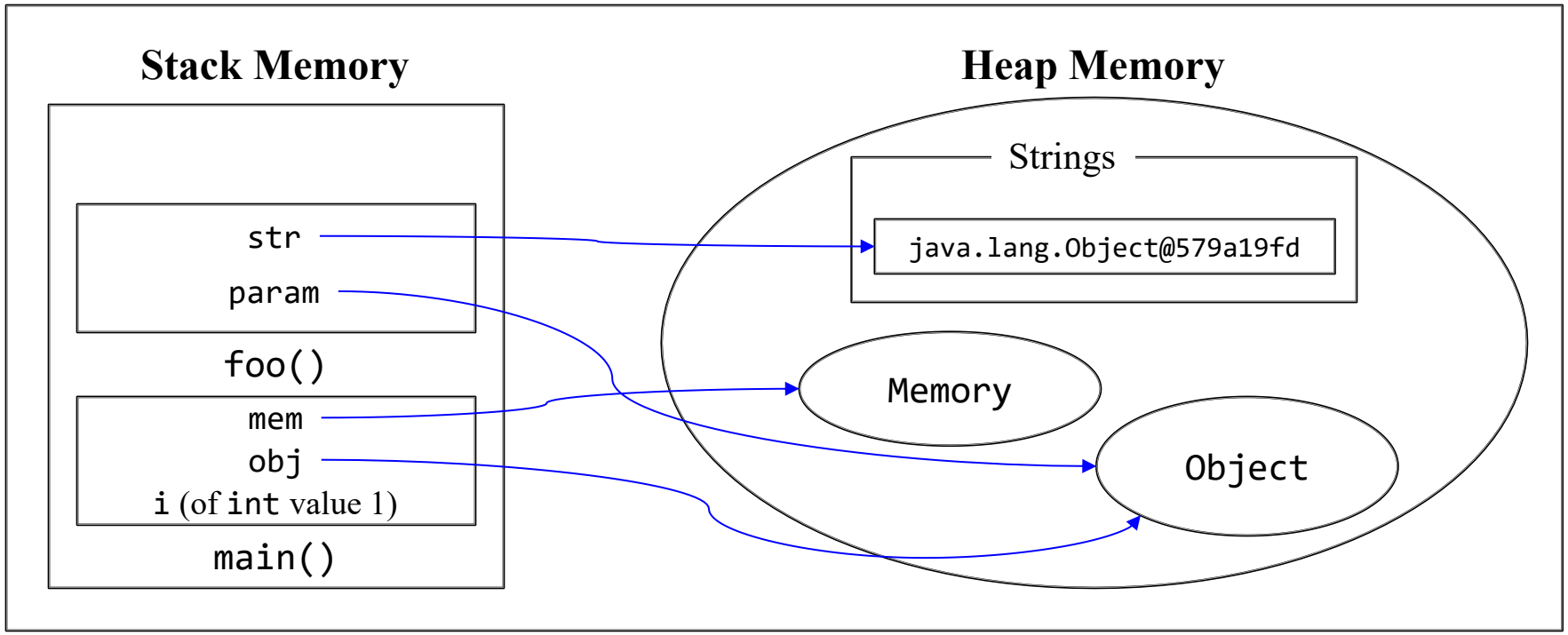
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

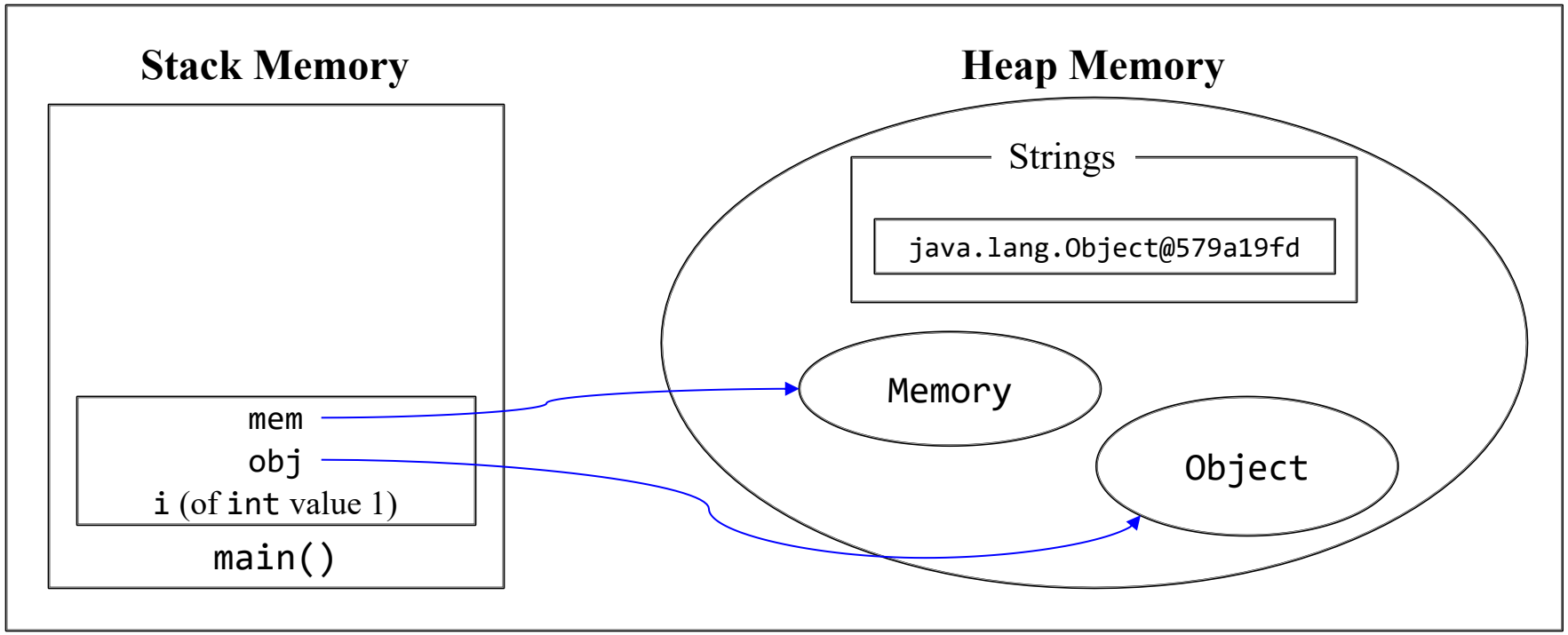
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

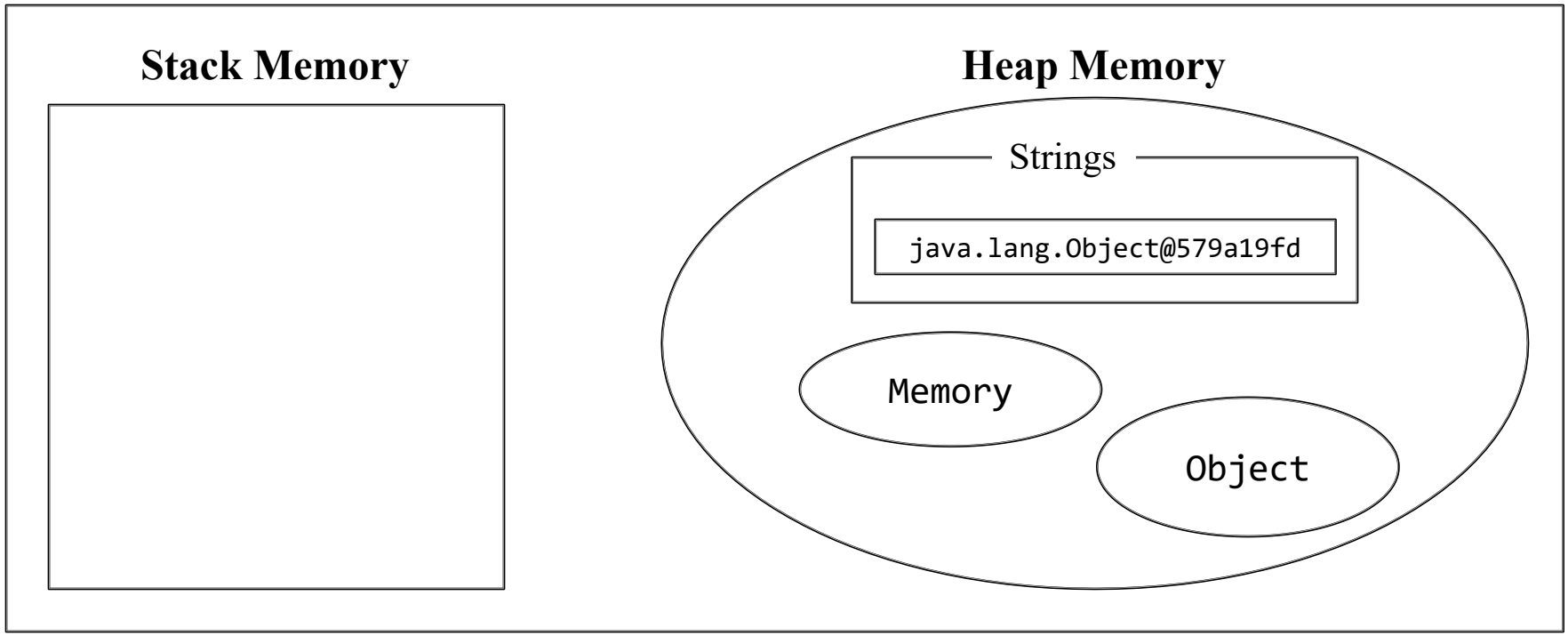
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



Java Runtime Memory

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

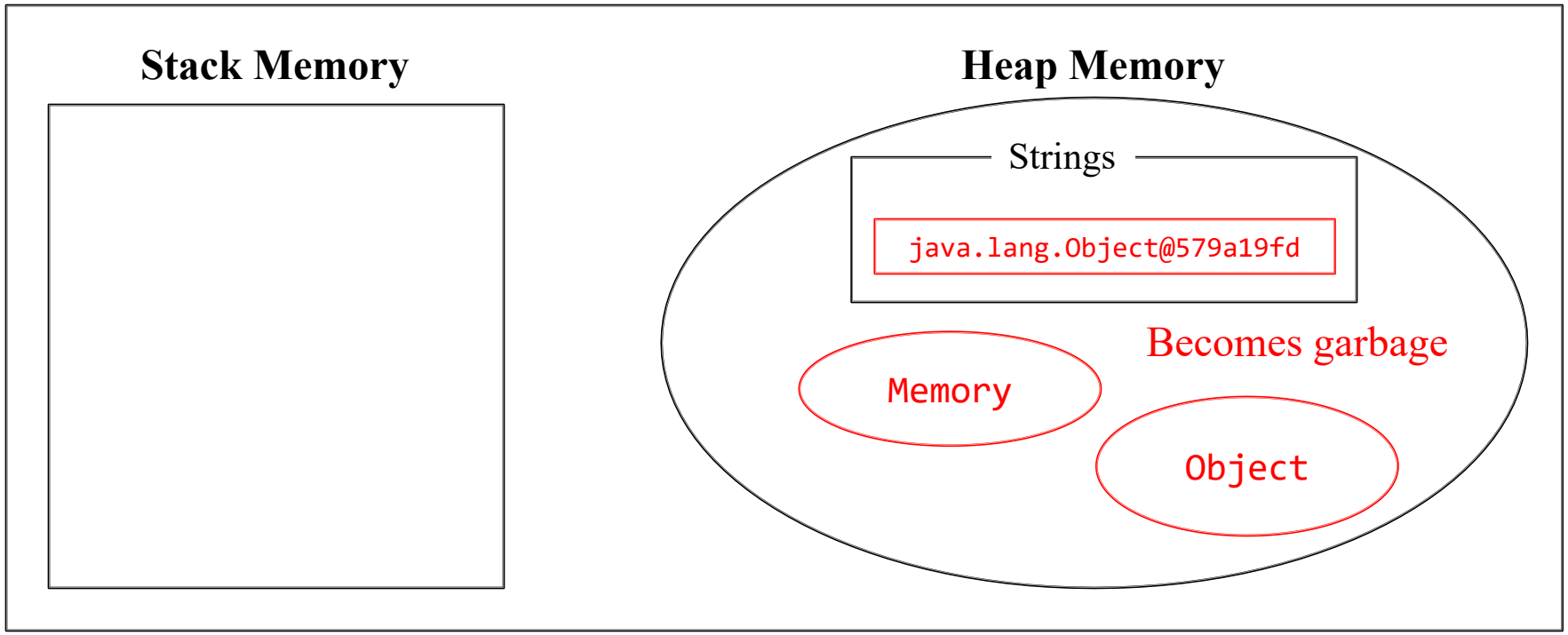
```
private void foo(Object param) {
    String str = param.toString();
    System.out.println(str);
}
```



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