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

Xuetao Wei

weixt@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

▶ 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.

```java
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

```java
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

```java
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

```java
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          Access the same variable
via Employee.getCount(): 2

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

`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("!");
```

Becomes garbage
if not referenced

| s1 | → ✖ → | Hello World |

| Hello World! |
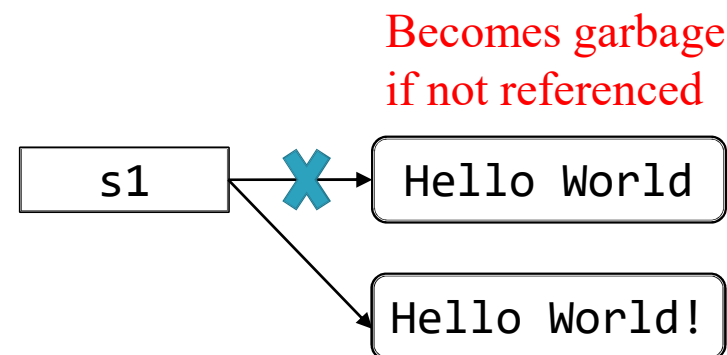
Garbage Collection

▸ 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;`

```java
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

```java
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

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;
No
modifier    int second;
            void setTime(int h, int m, int s) {...}
}
```

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)

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

```java
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);
    }

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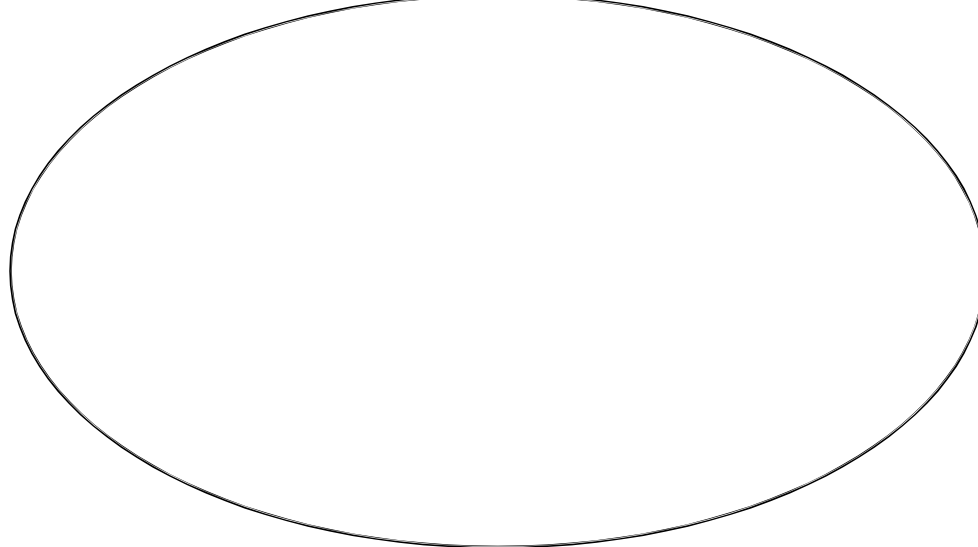
```java
public static void main(String[] args) {
    int i = 1;
    Object obj = new Object();
    Memory mem = new Memory();
    mem.foo(obj);
}
```

```java
private void foo(Object param) {

    String str = param.toString();

    System.out.println(str);

}
```
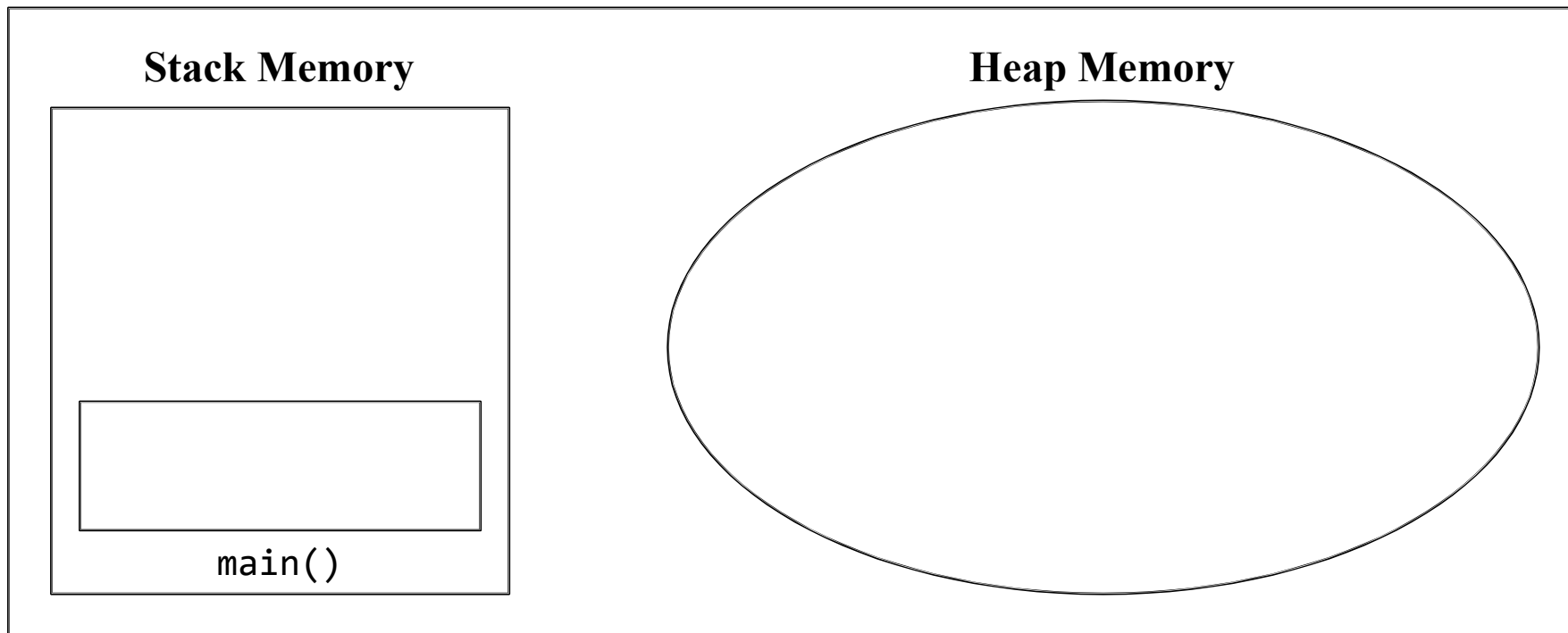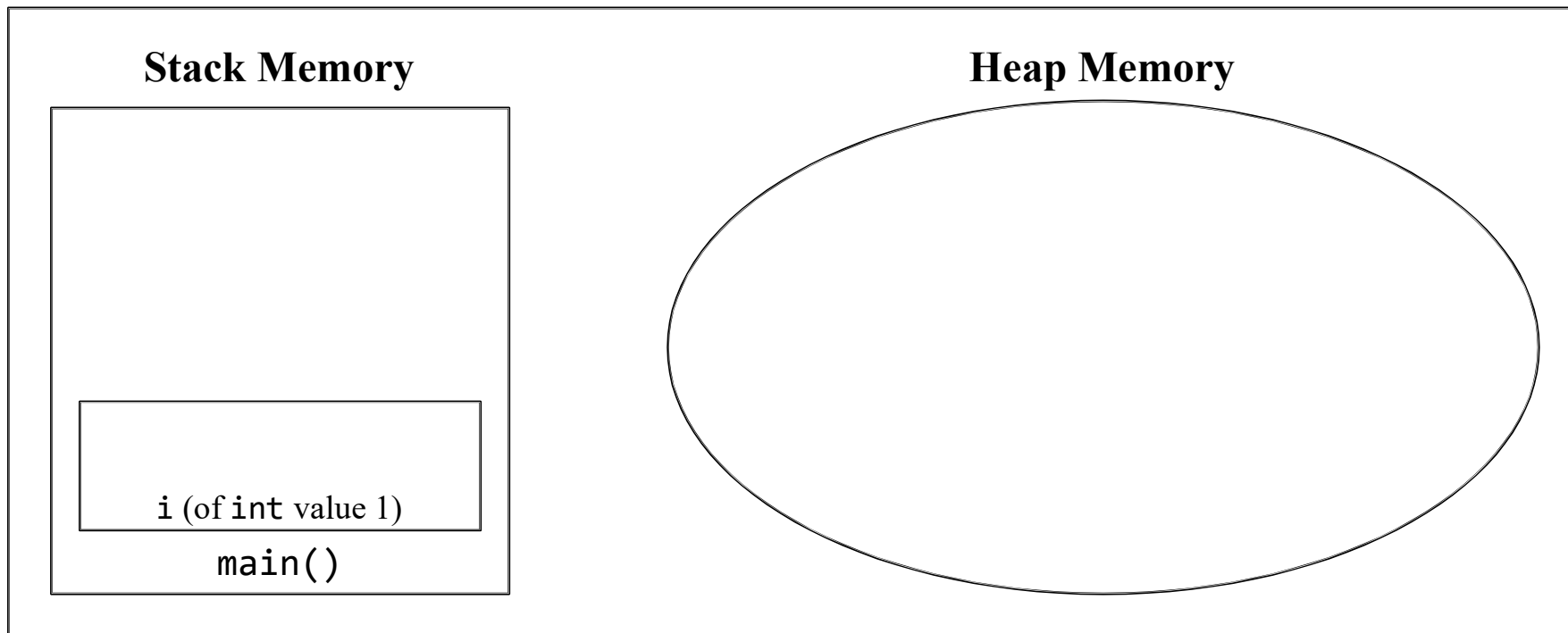
Stack Memory

Heap Memory

Java Runtime Memory

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

```java
private void foo(Object param) {

    String str = param.toString();

    System.out.println(str);

}
```

Stack Memory

Heap Memory

main()

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);

}
```
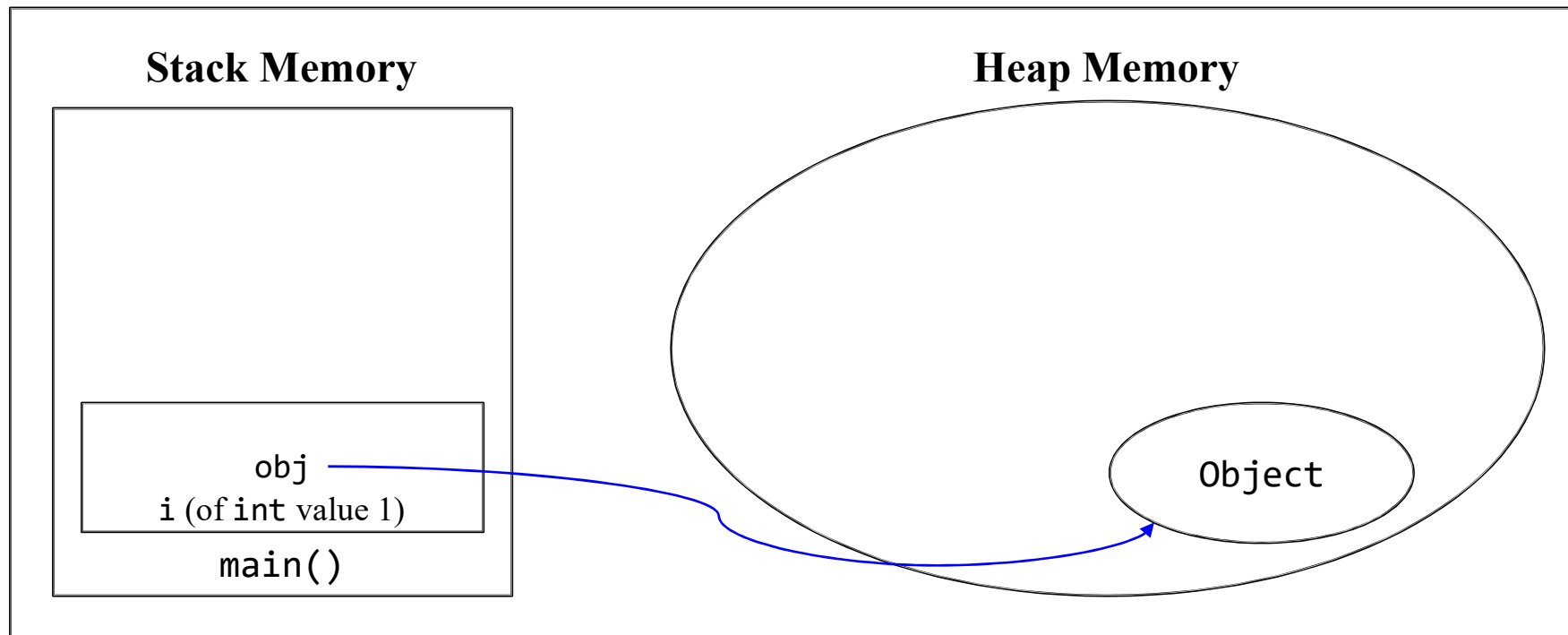
Stack Memory

Heap Memory

i (of int value 1)

`main()`

Java Runtime Memory

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

```java
private void foo(Object param) {

    String str = param.toString();

    System.out.println(str);

}
```
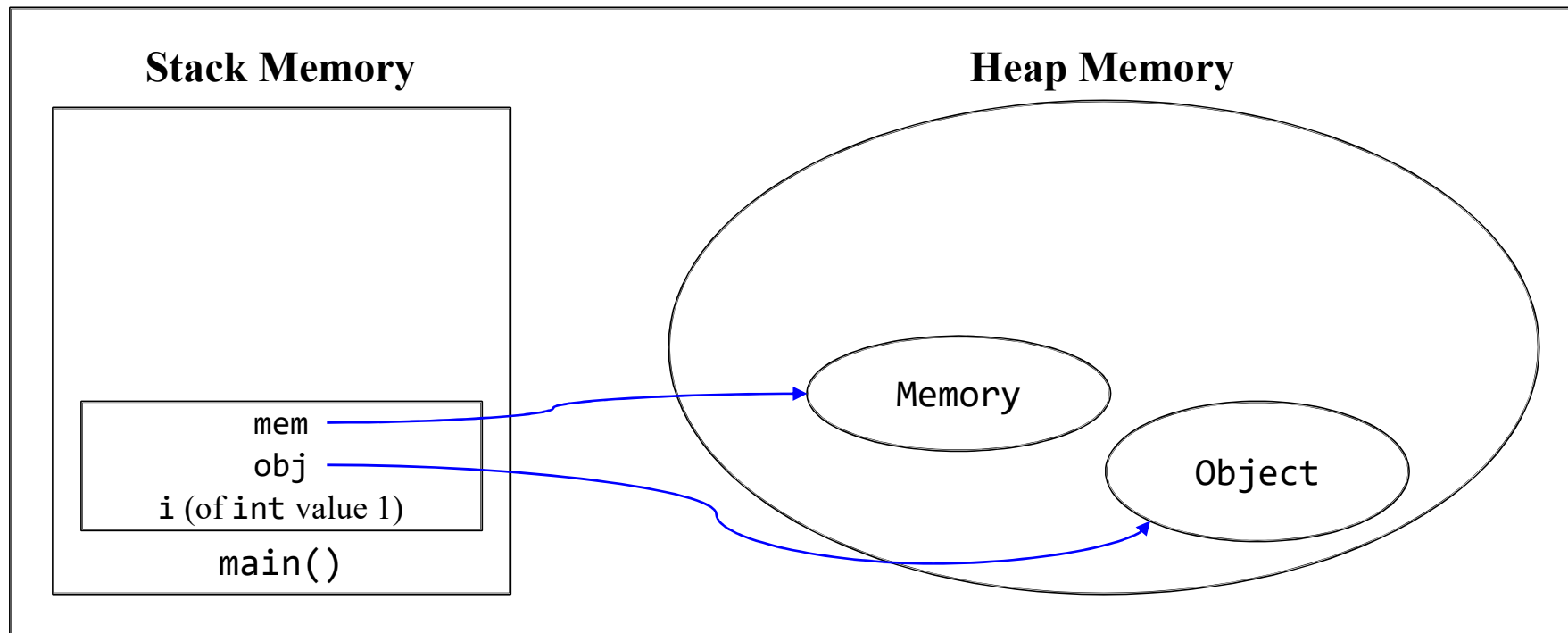
Stack Memory

Heap Memory

obj

i (of int value 1)

main()

Object

Java Runtime Memory

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

```java
private void foo(Object param) {

    String str = param.toString();

    System.out.println(str);

}
```
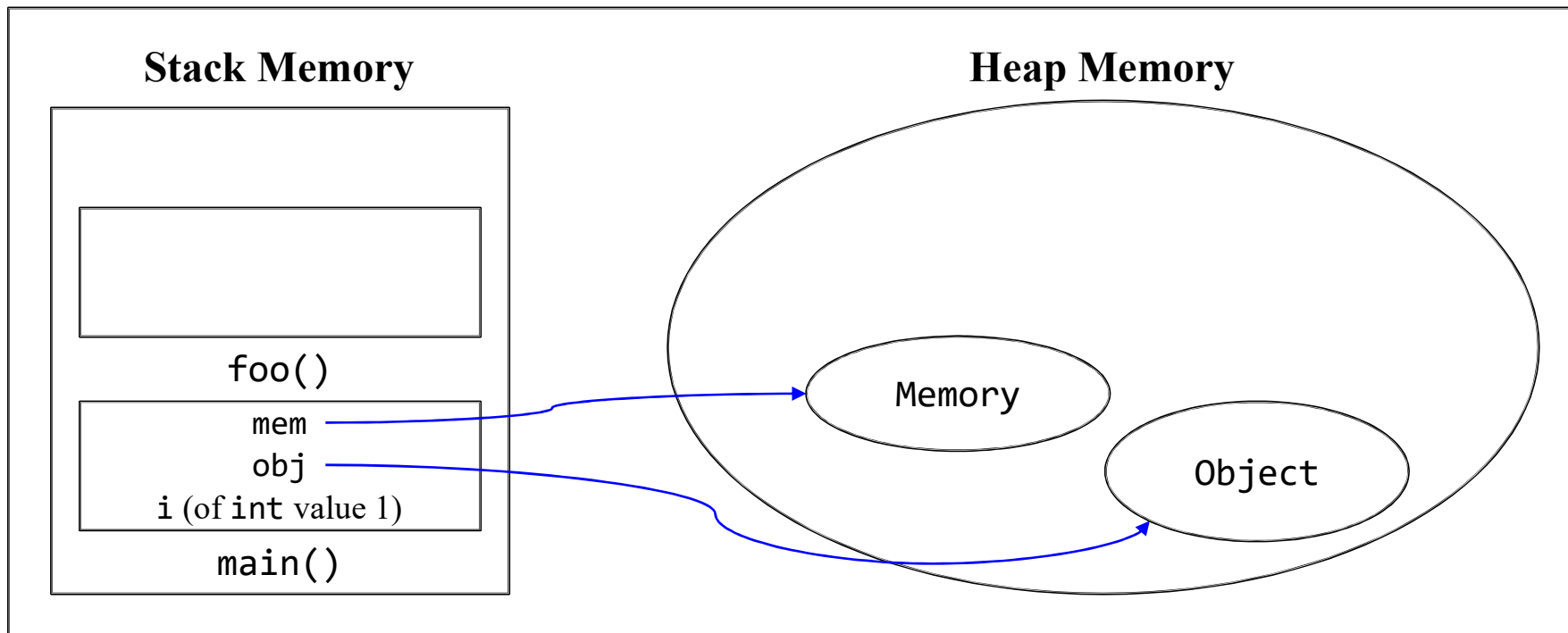
Stack Memory

Heap Memory

mem

obj

i (of int value 1)

main()

Memory

Object

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);

}
```
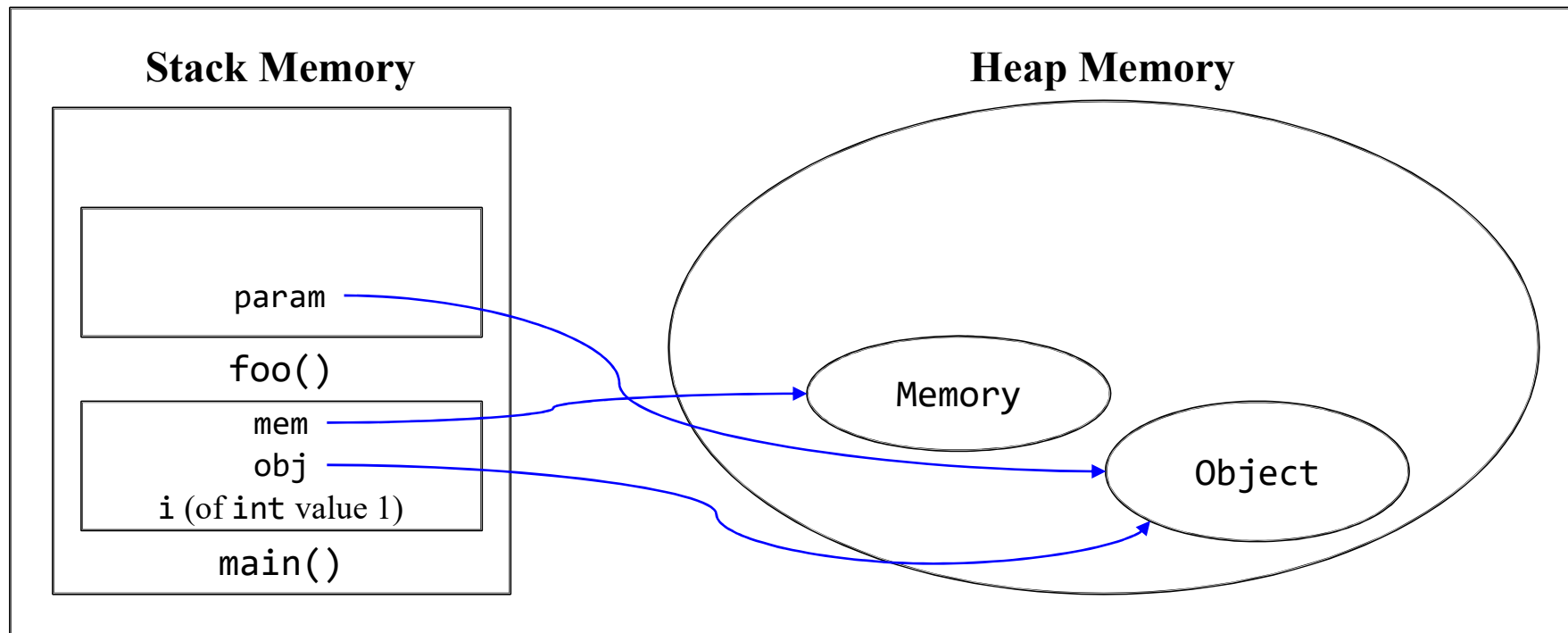
Stack Memory

Heap Memory

foo()

mem

obj

i (of int value 1)

main()

Memory

Object

Java Runtime Memory

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

```java
private void foo(Object param) {

    String str = param.toString();

    System.out.println(str);

}
```
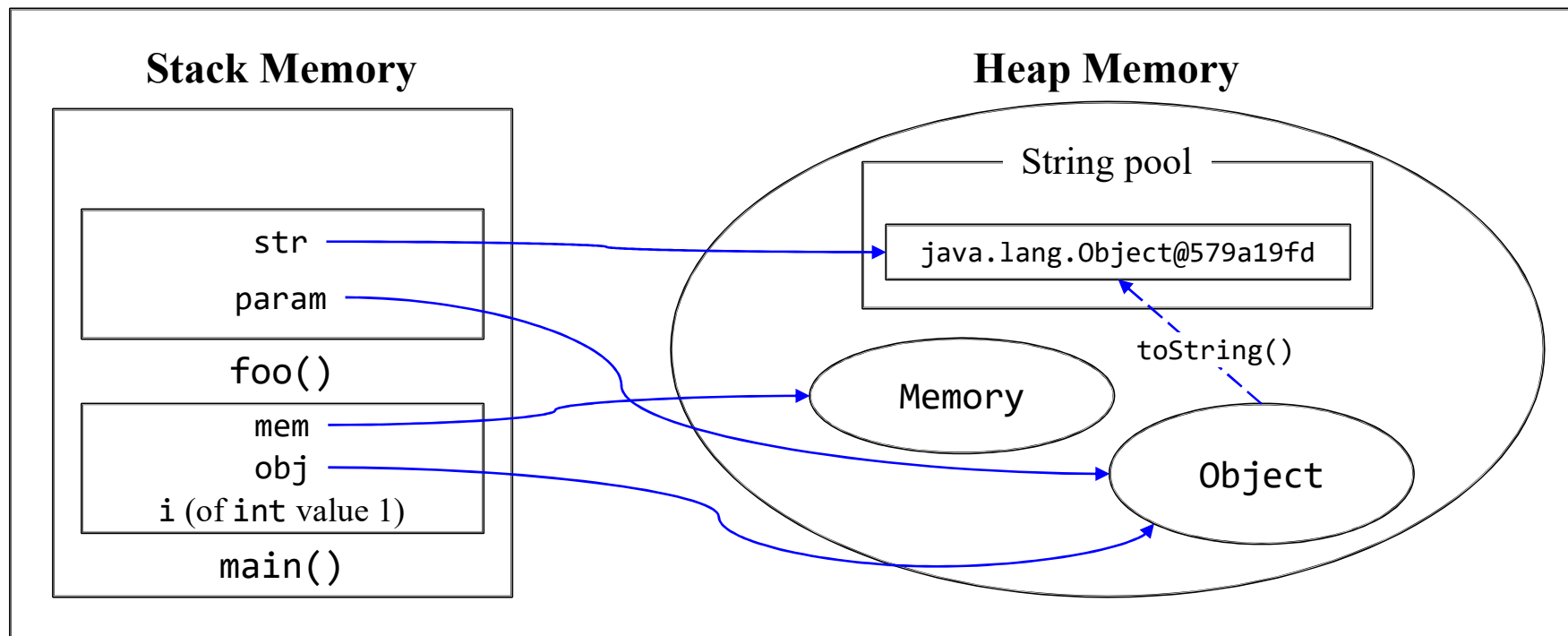

Java Runtime Memory

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

```java
private void foo(Object param) {

    String str = param.toString();

    System.out.println(str);

}
```


Stack Memory   **Heap Memory**

String pool

java.lang.Object@579a19fd

toString()

str

param

foo()

mem

obj

i (of int value 1)

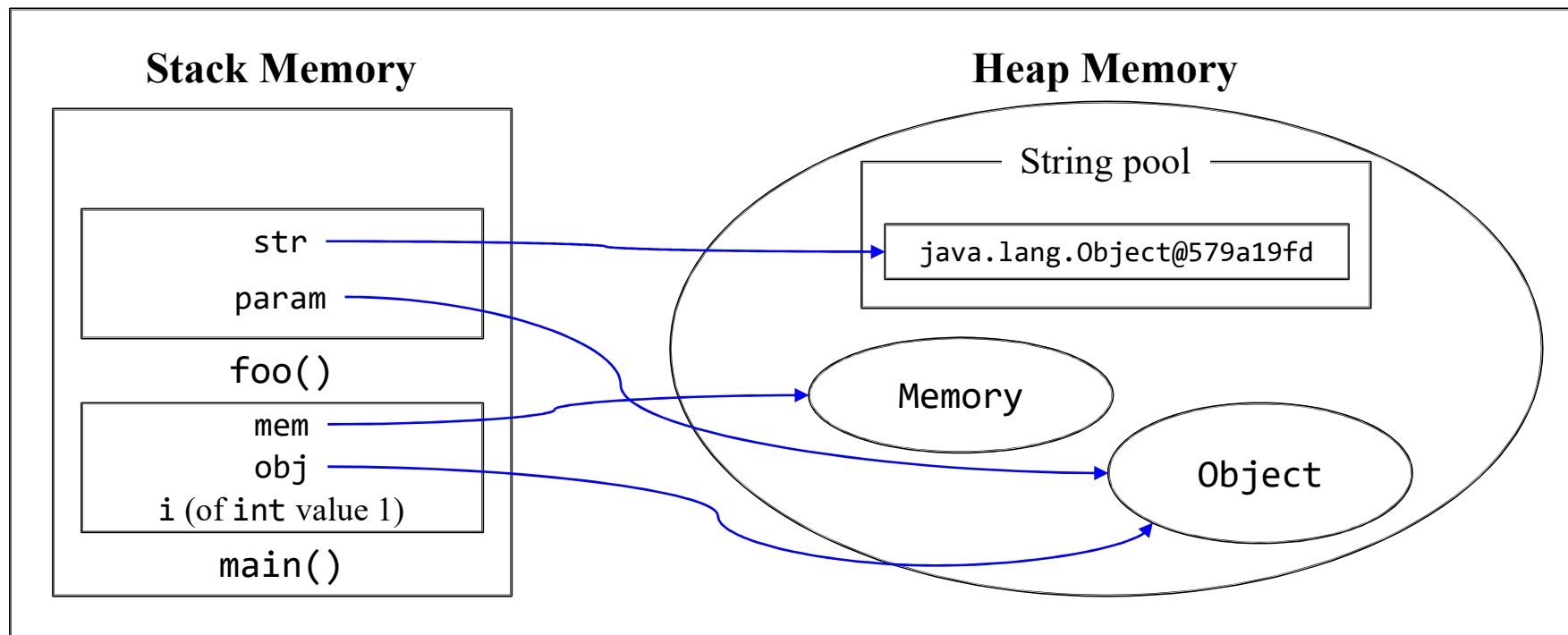main()

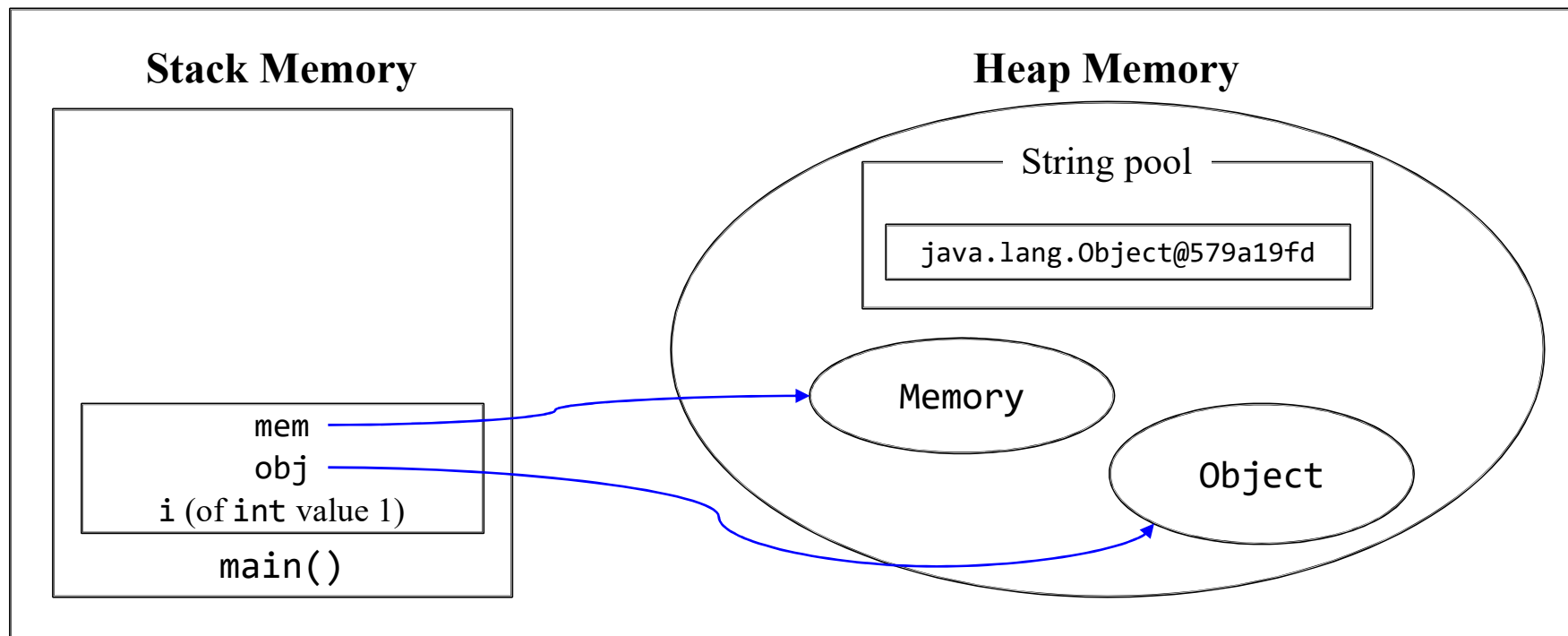Memory

Object

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

35

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

```java
private void foo(Object param) {

    String str = param.toString();

    System.out.println(str);

}
```

Stack Memory

Heap Memory

String pool

java.lang.Object@579a19fd

Memory

mem
obj
i (of int value 1)

Object

main()

Java Runtime Memory

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

```java
private void foo(Object param) {

    String str = param.toString();

    System.out.println(str);

}
```
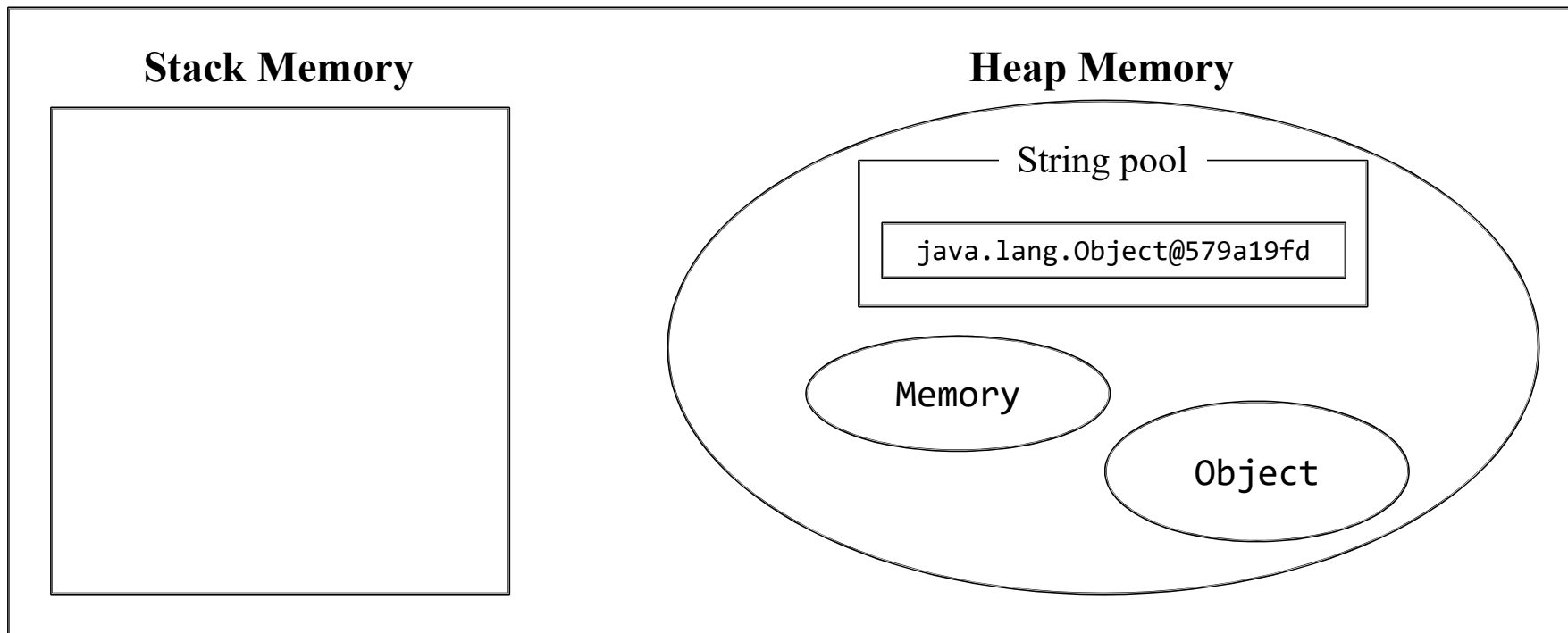
Stack Memory

Heap Memory

String pool

java.lang.Object@579a19fd

Memory

Object

Java Runtime Memory

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

```java
private void foo(Object param) {

    String str = param.toString();

    System.out.println(str);

}
```
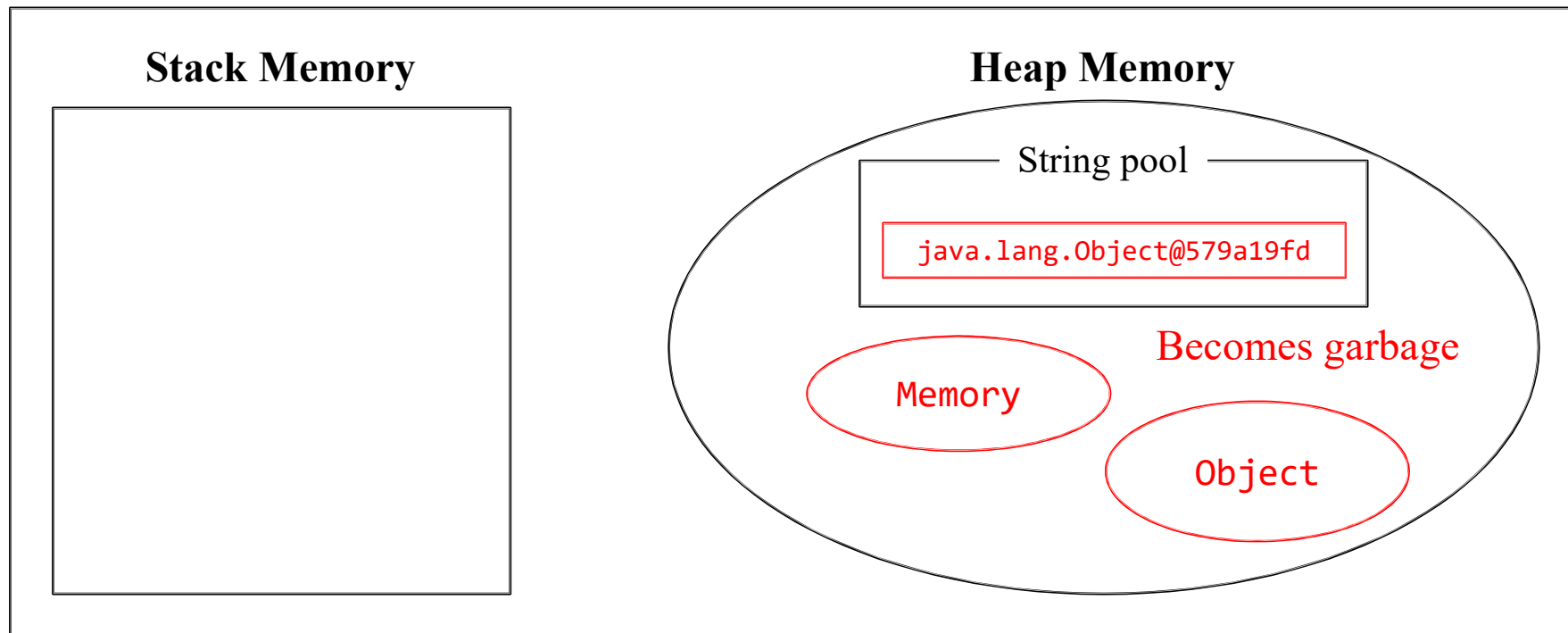
Stack Memory

Heap Memory

String pool

java.lang.Object@579a19fd

Becomes garbage

Memory

Object

Java Runtime Memory