Java run-time data storage

Sebastian Mateos Nicolajsen and Peter Sestoft
Computer Science Department, IT University of Copenhagen, Denmark

Version 1.0 of 26 January 2023

Abstract

We use simple drawings, snapshots of the computer memory, to show how data in Java are stored at run-time. The purpose is to make it clear how Java assignment, arrays, object creation, subclasses, conversions, casts, method calls, garbage collection, and so on work; in other words, what does a Java program do when it is run.

Such drawings are intended to be used both by educators to describe what Java's language constructs mean, and by students to hand in solutions to exercises about the meaning of Java programs.

An appendix shows how the descriptions apply to the C# language, with the necessary changes compared to Java.

1 Plan for this document

We start out with simple cases, such as primitive values of integer and floating-point types, then simple uses of strings, arrays and objects, later add more complex and subtle features of Java, and how they interact. We draw performance lessons, about both time and memory consumption from these. Hence these notes should be usable and of interest to students from the first semester right through graduation, and also to Java developers. Thus, these notes may be used for students wanting an overview of the way in which Java stores data as an example of a programming language implementation. The intention is not to give a complete description of how Java is implemented, or how the Java virtual machine works, but to give just enough details to understand how the Java *programming language* works: what is the effect and meaning of each Java language construct, what is its time and memory consumption. Thus, we reduce the Java memory model to be the stack(s) and heap — excluding other components such as the method area. We encourage curious and experienced individuals to further explore the components which we do not detail here.

In the rest of the document we discuss one aspect of Java on each page, using a Java program fragment and a drawing of the computer memory after executing that fragment.

We encourage the use of the "Python tutor" tool (or similar) to experiment with the various examples. We also intend to use the Java mode of the online

"Python tutor" tool (or similar) to permit interactive experimentation with these program fragments; a future version of this document may contain direct links to such examples.

Advanced Aspect. Most of the description of Java is valid for the C# programming language also, but with some variations and further aspects; see Appendix A.

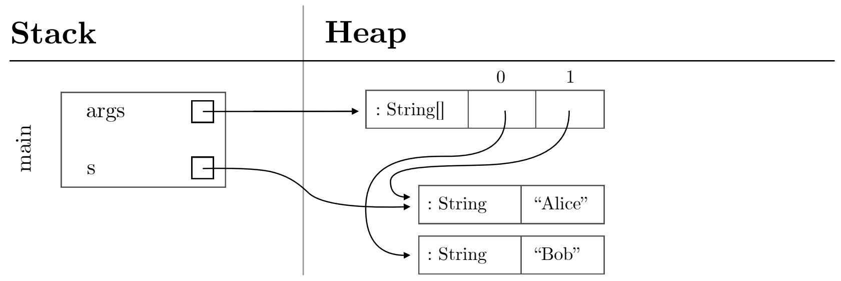
DISCLAIMER for this version: The Java code may contain mistakes.

2 The stack and the heap

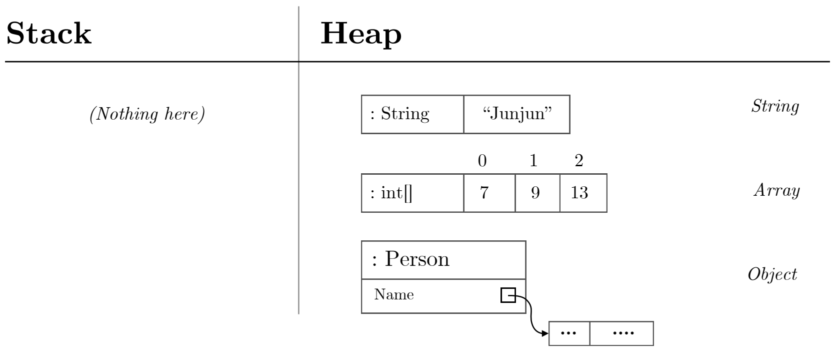
The Java run-time memory holds the values of variables, parameters, object fields and the like during a Java program's execution. The run-time memory is divided into the stack and the heap. The *stack* stores the values of local variables and parameters of all currently executing method calls. The *heap* stores all objects (that is, instances of classes) including all the objects' fields. Note that strings, arrays, collections, file handles, closures of anonymous functions (lambdas) and so on are also objects, and therefore stored in the heap.

Both the stack and the heap are stored in the computer's RAM, or random access memory.

A variable in the stack may refer to a value (which is an object) in the heap. This is shown by an arrow that points from the stack to the value in the heap. Likewise, an array element or an object field in the heap may refer to a value in the heap, similarly shown by an arrow that points from inside the array or object to a value in the heap. There are never references (arrows) from the heap to the stack, and never references (arrows) that point to something *inside* a value (object, string, array, lambda closure, ...) in the heap, only to the value as a whole. Here we show a snapshot of a stack with a single frame containing parameter `args` and variable `s`, and a heap containing an array of strings, referred to by `args`, and two strings `"Alice"` and `"Bob"`, referred to from the array elements and from the variable `s`:



Each object in the heap has a *header* describing the value's run-time type, followed by the value's contents. For strings and arrays we draw the header and contents on the same line; for other objects we draw the object's fields below the header:



Advanced Aspect. The stack is divided into frames. When a method call `m(...)` is executed, a new frame is added to the top of the stack to hold the values of the variables and parameters of method `m`. When the method call returns, the top frame is removed from the stack, and those variables and parameters are no longer in the stack. Until Section 3.15 we will consider only stacks with a single frame, namely the frame holding the `main` method's variables and parameters.

Advanced Aspect. The run-time type in an object's header is used during Java program execution in virtual method calls (Section 3.19), reference-type casts (Section 3.12), unboxing operations (Section 3.13), array element assignments (Section 3.8), exception catching (Section 3.22) and during garbage collection (Section 3.25).

Advanced Aspect. In a multi-threaded Java program, there is a separate stack for each executing thread but there is only one heap, shared between all threads. Until Section 3.26 we will consider only programs with a single thread, namely the initial thread that calls the `main` method. In a multi-threaded program, threads can communicate and collaborate with each other only through reading and writing data in the shared heap.

2.1 Value types and reference types

A *value type* in Java is a primitive type, such as an integer type, floating-point type, or `boolean`. Variables of value type are stored directly in the stack. Likewise, array elements and object fields of value type are stored directly inside the array or object (which is stored in the heap).

A *reference type* in Java is String, an array type, a class, an interface, or a generic parameter T as in Set<T>. A variable of reference type is either `null` and refers to nothing, or is non-`null` and refers to an object stored in the heap. Hence, the stack cannot actually contain a reference-type value; instead it contains a reference to the reference-type value, which is in the heap.

Advanced Aspect. C# has additional value types, namely structs and tuples; see Appendix A.

3 Java features and resulting data storage

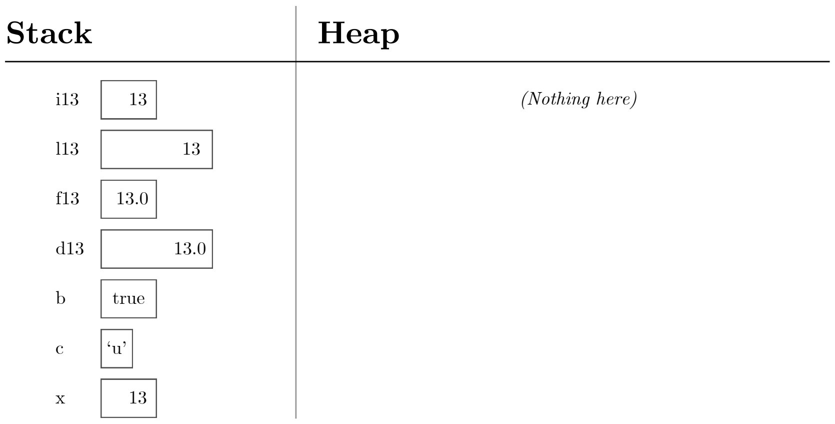
We always show the stack on the left and the heap on the right. Each object in the heap is shown as a box with the object's run-time type first, followed by the object's contents. The run-time type may be String, String`[]`, Person, Person`[]`, and the like. The object's contents is the characters of the string, the elements of the array, the instance fields of the object, and so on.

3.1 Primitive types

Primitive value types include the integer number types such as `int` and `long`, the floating-point types `float` and `double`, the truth-value type `boolean`, and the character type `char`. Values of variables of these types are stored directly in the stack, and assignment such as `x = i13` will copy the values:

```java
int i13 = 13;
long l13 = 13;
float f13 = 13.0;
double d13 = 13.0;
boolean b = true;
char c = 'U';
int x = i13;
```

Running the above Java code produces the following stack and heap contents:



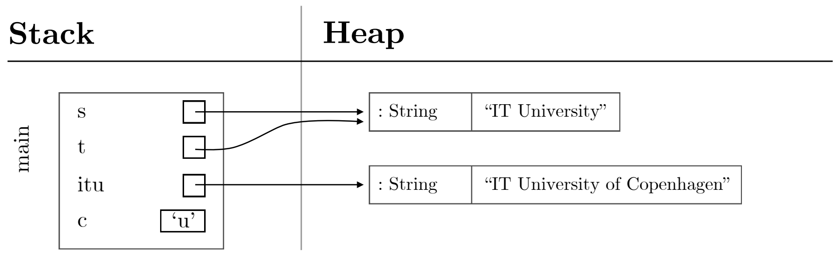
3.2 Strings

The String type is a reference type, so a string value is stored in the heap. A string value consist of the type String, followed by the characters of the string. A variable of type String in the stack refers to a String object in the heap. Assignment of a string value to a variable of type String copies only the reference, not the string value. A string value is immutable (that is, cannot be updated), so evaluation of an expression of type String typically creates a new string object in the heap.

Advanced Aspect. While the String type is a reference type, one can utilise the *String Constant Pool* of Java, a particular area of the heap, to allow comparison and reuse of identical strings. Literals are automatically stored here, and generated strings can be stored using the method *intern*.

```java
String s = "IT University";
String t = s;
String itu = s + " of Copenhagen";
char c = itu.charAt(3);
```

Running the above Java code produces the following stack and heap contents:



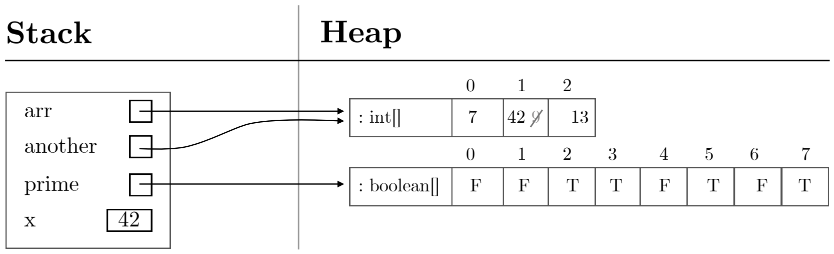
3.3 Arrays of primitive type

Every array type is a reference type, so an array value is stored in the heap. An array value consists of a header describing the array's run-time type, followed by the array's elements, with indexes 0, 1, 2, ... A variable of array type contains a reference to the array value in the heap. Assignment of an array value to a variable of array type copies only the reference, not the array value. The elements of an array are mutable (that is, can be updated). Assignment `arr[1] = 42` to an array element changes only that array element; it does not create a new array value.

When the array elements are of primitive type, such as `int`, the values are stored directly inside the array elements.

```java
int[] arr = new int[] { 7, 9, 13 };
int[] another = arr;
arr[1] = 42;
boolean[] prime =
  new boolean[] { false, false, true, true, false, true, false,
    true};
int x = another[1];
```

Running the above Java code produces the following stack and heap contents:



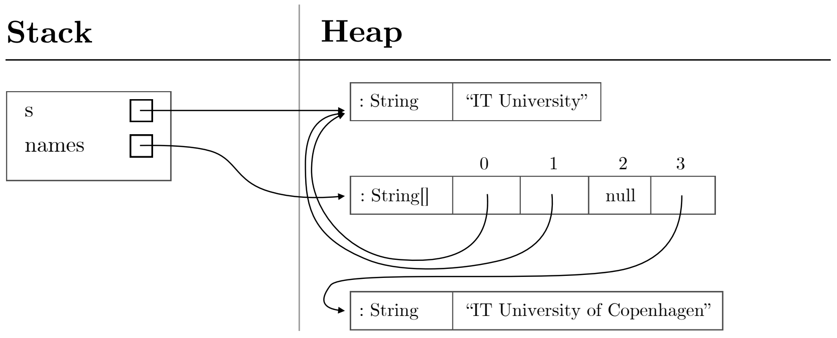
3.4 Arrays of strings

An array of strings has type String[], which is a reference type like all other array types. Since String is a reference type, assignment `names[1] = s` to an array element just copies the reference to the string value `s`, it does not copy the string value.

When the array elements are of reference type, such as String, only a reference to the value is stored in each array element, and all array elements initially hold the value `null`.

```java
String s = "IT University";
String[] names = new String[4];
names[0] = s;
names[1] = s;
names[3] = s + " of Copenhagen";
```

Running the above Java code produces the following stack and heap contents:



3.5 Classes and objects

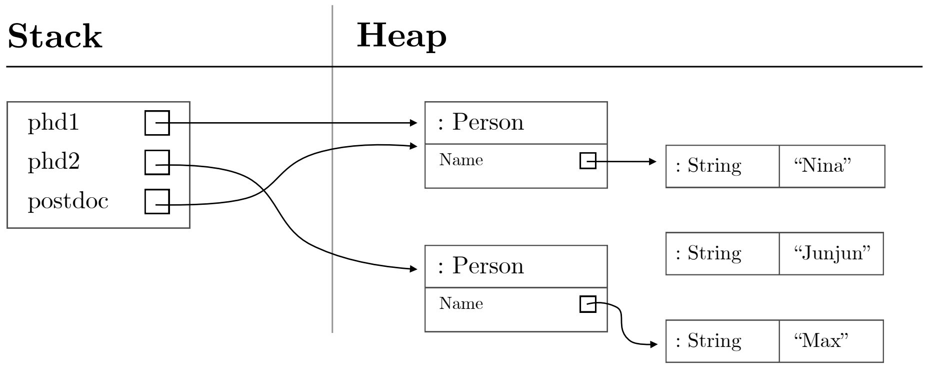
An object is a value; it is an instance of a class, created with `new`, it is a reference type, and is stored in the heap. An object consists of a header describing the object's run-time type, followed by the object's instance fields. A variable of class or interface type contains a reference to an object in the heap, or is `null`. Assignment of an object value to a variable of class or interface type copies only the reference, not the object. The fields of an object are mutable (that is, can be updated), unless declared `final`. Assignment `phd2.name = "Max"` to a field of an object changes only that field; it does not create a new object.

When a field of an object has reference type, such as String, the field stores only a reference to its value, not a copy of the value.

Inside an object's instance methods and constructors, the keyword `this` is a reference to the object itself. It is never `null`.

```java
class Person {
  public String name;
  public Person(String name) { this.name = name; }
}
Person phd1 = new Person("Nina");
Person phd2 = new Person("Junjun");
Person postdoc = phd1;
phd2.name = "Max";
```

Running the above Java code produces the following stack and heap contents:



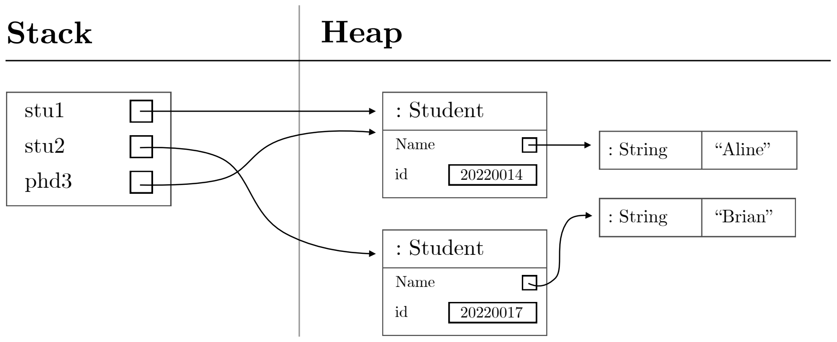
3.6 Subclasses and objects

An object of a subclass is an instance of a class that is a subclass of another class. An object value consists of a header describing the object's run-time type, which is the subclass, followed by the object's instance fields; these are all the instance fields of the base class, and possibly more. A reference to an instance of a subclass can be assigned to a variable whose type is that subclass, or any base class or base interface.

Continuation of the example in Section 3.5:

```java
class Student extends Person {
  public int id;
  public Student(String name, int id) { super(name); this.id =
    id; }
}
Student stu1 = new Student("Aline", 20220014);
Student stu2 = new Student("Brian", 20220017);
Person phd3 = stu1;
```

Running the above Java code produces the following stack and heap contents:



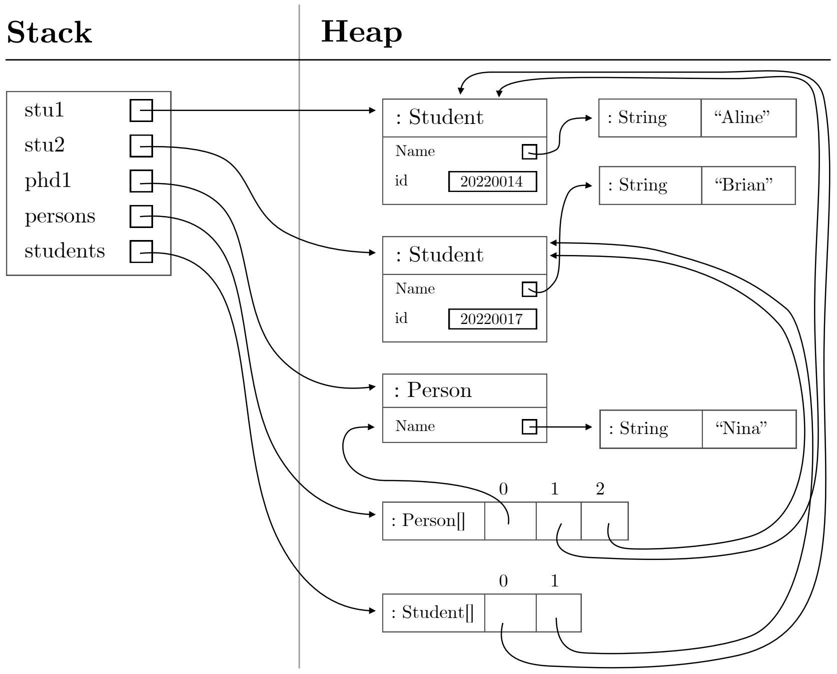
3.7 Subclasses and arrays of objects

An array of Person objects has run-time type Person`[]`. Each element must contain a reference to an object whose run-time type is Person or one of its subclasses, or contain `null`. Since Student is a subclass of Person, an array of Person objects can contain a reference to a Student object.

Continuation of the examples in Sections 3.5 and 3.6:

```java
Person[] persons = new Person[] { phd1, stu1, stu2 }
Student[] students = new Student[] { stu1, stu2 };
```

Running the above Java code produces the following stack and heap contents:



3.8 Advanced Aspect: Array type covariance and array element assignment

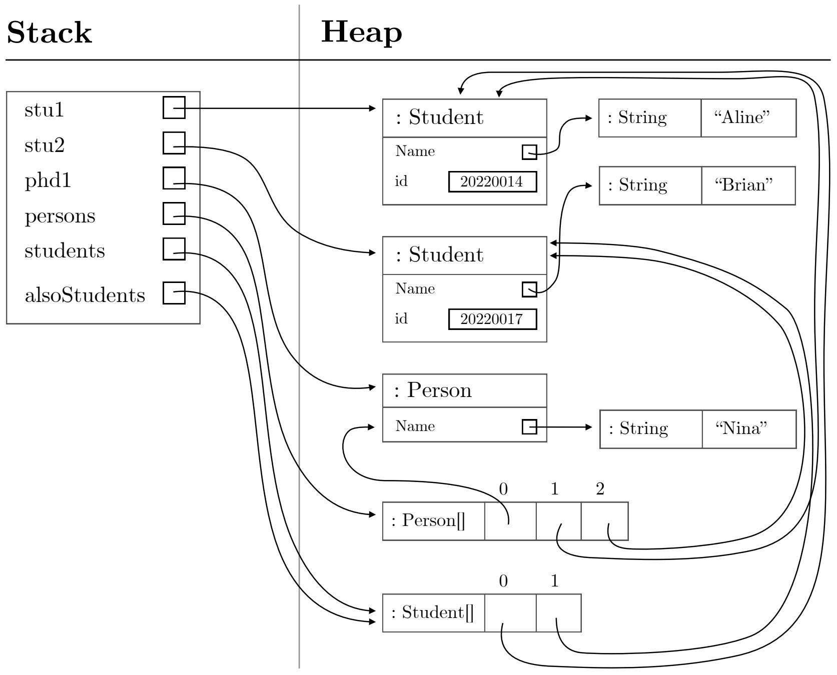
Since Student is a subclass of Person, also the array type Student`[]` is considered a subtype of the array type Person`[]`, so the assignment `alsoStudents = students` is allowed. This is called array type covariance.

Due to array type covariance, each assignment to an element of an array of reference type must check that the run-time type of object to be assigned is a subtype of the array's run-time element type. These run-time types are found in the array header and the object header. The assignment `alsoPersons[0] = stu1` is allowed because the run-time type of the array referred to by `alsoPersons` is Student`[]`, and the run-time type of `stu1` is Student. On the other hand, the assignment `alsoPersons[0] = phd1` would throw an exception at run-time because the run-time type of the value of `phd1` is Person, which is not a subtype of Student, the run-time element type of `alsoPersons`. The compiler does not discover this, because the compile-time type of `alsoPersons` is Person`[]` and so it believes that the assignment is legal.

Continuation of the examples in Sections 3.5, 3.6 and 3.7:

```java
Person[] alsoStudents = students;
alsoStudents[0] = stu1;          // Compiles OK; works at run-
    time
// alsoStudents[0] = phd1;       // Compiles OK; throws at run-
    time
```

Running the above Java code produces the following stack and heap contents:



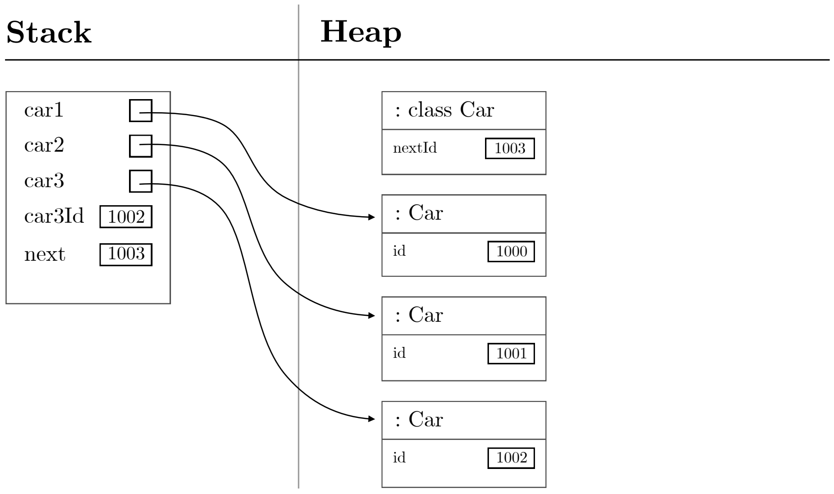
3.9 Static fields in classes

An object instance of a class holds only the class's instance fields — those fields not declared `static`. In the heap there will be as many copies of each instance fields as there are objects of the class, but there will be exactly one copy of each static field, stored in a special object that represents the class at run-time. Below, three objects of class Car are created, each with a distinct value of the `id` instance field. But even before the first object is created, an automatically created object holds the single copy of the static field `nextId`.

Advanced Aspect. While the class definition and static fields of a class are stored on the heap, it is so in a particular area named the *class(method) area*.

```
class Car {
  public static int nextId = 1000;
  public int id;
  public Car() {
    this.id = nextId;
    nextId = nextId + 1;
  }
}
Student Car car1 = new Car(), car2 = new Car(), car3 = new Car()
    ;
int car3Id = car3.id;
int next = Car.nextId;
```

Running the above Java code produces the following stack and heap contents:



3.10 Primitive-type conversions and casts

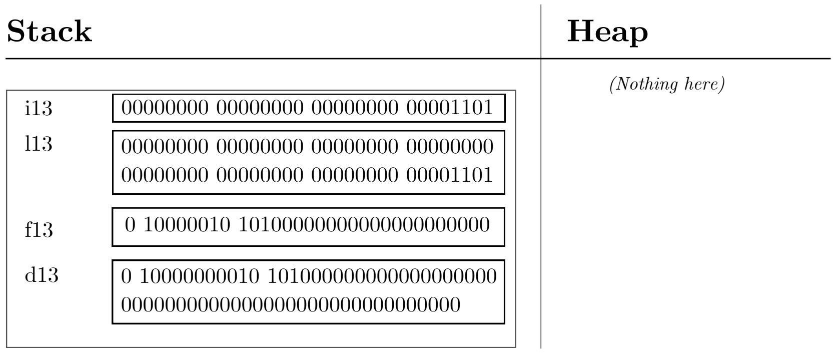
A conversion of a primitive type value to another primitive type, for instance of integer 13 to a `float`, typically produces a new bit pattern to represent the value. Primitive type conversions may be narrowing (and possibly change the value), or widening (and preserve the value), or may be lossy (and preserve the value but cause some loss of precision); see [JP, §5.7]. A narrowing conversion must be written with an explicit cast such as `(int)13.7`, which truncates the floating-point number 13.7 to the integer 13.

Section 3.12 describes reference type casts such as `(Student)r` which look exactly like primitive-type conversions, but never change the reference `r`. Section 3.13 describes unboxing and boxing such as `(Integer)i13` which look exactly the same, but convert between primitive values and objects.

The primitive types that can be converted to each other are `char`, `byte`, `short`, `int`, `long`, `float` and `double`; there is no conversion to or from `boolean`.

```java
int i13 = 13;
long l13 = i13;          // Widening conv., result 13
float f13 = i13;         // Lossy conv., result 13.0 (no actual
    loss)
double d13 = i13;        // Widening conv., result 13.0
```

Running the above Java code produces the stack contents below (and an empty heap). We use the actual bit patterns to show that the number 13 is represented in four different ways: integer versus floating-point, and 32-bit (`int`, `float`) versus 64-bit (`long`, `double`). See [CA] and [DG] for more information about computer number representations.



Here is an example showing a conversion (to `float`) that loses precision, and one that does not:

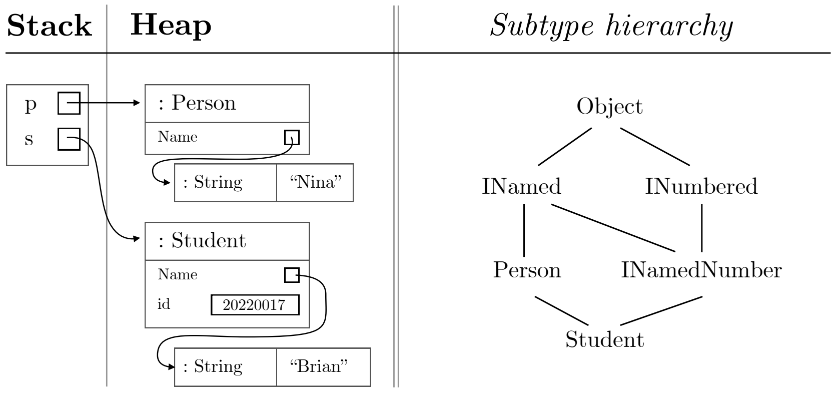
```java
float f = 1000111222;    // Lossy, f is 1.00011123E9 = 1000111230
double d = 1000111222;   // Widening, d is 1.000111222E9 =
    1000111222
```

3.11 Instance-of tests

An instance-of test has the form `(r instanceof C)` where `C` is a class or interface and `r` has reference type. It evaluates to true if `r` is non-`null` and its run-time type, found in the object header in the heap, is `C` or a subtype of `C`. Otherwise the test returns false; in particular if `r` is `null`.

```java
interface INamed { String getName(); }
interface INumbered { int getId(); }
interface INamedNumbered extends INamed, INumbered { }
class Person implements INamed {
  public String name;
  public Person(String name) { this.name = name; }
  public String getName() { return name; }
}
class Student extends Person implements INamedNumbered {
  public int id;
  public Student(String name, int id) { super(name); this.id =
    id; }
  public int getId() { return id; }
}
Object p = new Person("Nina"), s = new Student("Brian",
    20220017);
```

The class and interface declarations above create the subtype relations shown on the right below. Running the Java code produces the stack and heap contents shown on the left below; note that variables `p` and `s` of compile-time type Object point to values with run-time type Person and Student.



The following instance-of tests would all return true: `(p instanceof Person)` and `(s instanceof Student)` and `(s instanceof Person)` and `(p instanceof INamed)` and `(s instanceof INamed)` and `(s instanceof INumbered)` and `(s instanceof INamedNumbered)` and `(p instanceof Object)` and `(s instanceof Object)`. The following instance-of tests would all return false: `(null instanceof Object)` `(p instanceof Student)` and `(p instanceof INamedNumbered)` and `(p instanceof INumbered)`.

3.12 Reference-type casts

A reference-type cast has the form `(C)r` where `C` is a class or interface and `r` has reference type. The cast succeeds if `r` is `null`, or if `r` is non-`null` and its run-time type, found in the object header in the heap, is `C` or a subtype of `C`. Otherwise the cast fails by throwing a CastClassException. In fact, the cast `(C)r` will succeed if the test `(r instanceof C)` returns true or `r` is `null`; and will fail if the test `(r instanceof C)` returns false and `r` is not `null`.

A reference-type cast is a pure check: the run-time result of the expression is always `r` itself; unlike primitive-type conversions and boxing/unboxing operations it does not change any data.

Using the interface, class and variable declarations shown in Section 3.11, the following reference-type casts would succeed:

- `(Person)p`
- `(Student)s`
- `(Person)s`
- `(INamed)p`
- `(INamed)s`
- `(INumbered)s`
- `(INamedNumbered)s`
- `(Object)p`
- `(Object)s`
- `(Person)null` and all other reference-type casts of `null`

The following casts would fail, throwing ClassCastException:

- `(Student)p`
- `(INamedNumbered)p`
- `(INumbered)p`

3.13 Boxing and unboxing

Sometimes a primitive type value needs to be represented as an object in the heap. For instance, to store an `int` value in a collection Set<Integer>, it must be wrapped as an object of class Integer. There exist such a wrapper class (Boolean, Float, Double and so on) for each primitive type.

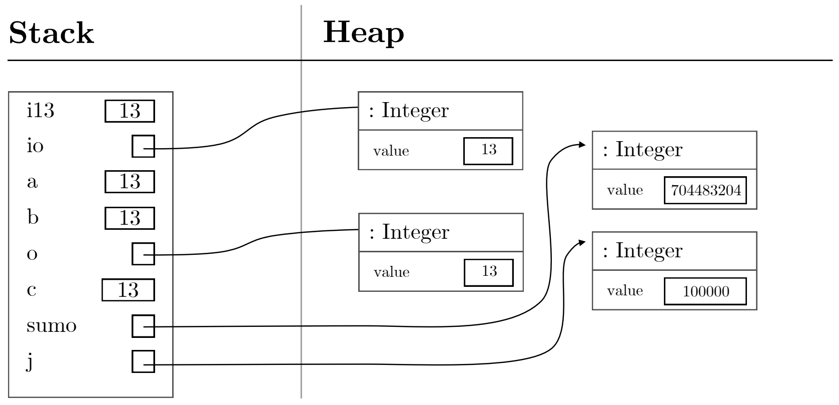
The conversion from primitive type to object is called *boxing*, and the opposite conversion is called *unboxing*. Boxing and unboxing are performed automatically when needed. Boxing and unboxing may also be performed explicitly using operations such as `Integer.valueOf(i)` to box the integer `i`, and `o.intValue()` or the cast `(int)o` to unbox the Integer object `o`. If `o` is `null`, then unboxing of `o` will fail at run-time by throwing NullPointerException.

Boxing requires creating a new object on the heap and then copying the value into a field of that object. Creating an object takes time, so avoid unnecessary boxing if performance is important. The for loop below would be 8 times faster if `sumo` and `j` had been declared as primitive type `int` instead.

Unboxing requires checking the run-time type of the object, and then copying the value out of the object; both are relatively fast operations.

```java
int i13 = 13;
Integer io = i13;
int a = io.intValue();
int b = (int)io;
Object o = i13;
int c = (int)o;
Integer sumo = 0;
for (Integer j=1000; j<100000; j++)
  sumo += j;
```

Running the above Java code produces the following stack and heap contents:



3.14 Closure of an anonymous function (lambda)

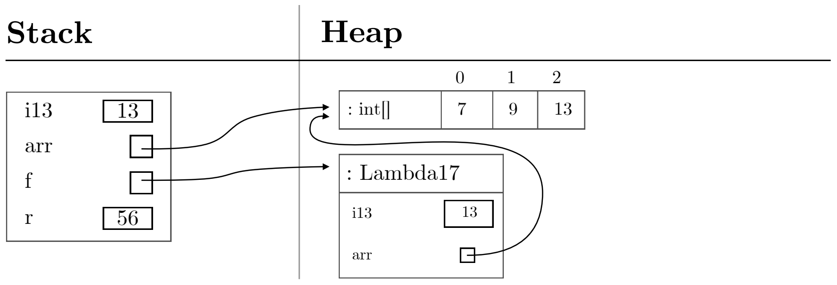
An anonymous function, or lambda expression, is an expression that evaluates to a so-called object. The body of a lambda may mention variables, such as `i13` or `arr`, that are defined outside the lambda; these are called captured variables. Any captured variable must be effectively final: it must not be updated neither outside nor inside the lambda.

The lambda expression evaluates to a closure object, whose fields hold copies of the values that the captured variables had when the lambda was created. Since the captured variables are effectively final, these values remain the same as those of the original variables.

Advanced Aspect. If you need the body of a lambda to update a variable, say variable `x` of type `int`, then declare instead a variable `final int[] xholder = new int[1]` outside the anonymous function, and everywhere use `xholder[0]` instead of `x`. The variable `xholder` on the stack will be effectively final, but the array element `xholder[0]` in the heap can still be updated.

```java
int i13 = 13;
int[] arr = new int[] { 7, 9, 13 };
Function<Integer,Integer> f =
  x -> { arr[2] = arr[2] + 10; return x + i13 + arr[2]; };
int r = f.apply(20);
// Now arr[2] is 23, and r is 20 + 13 + 23 = 56
```

Running the above Java code produces the following stack and heap contents:



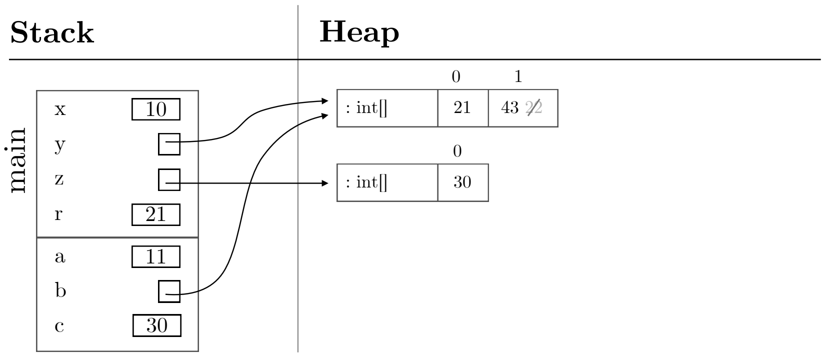
3.15 Static methods and parameter passing

A call `m(x+1, y, z[0])` to a static method `int m(int a, int[] b, int c)` creates a new stack frame for holding the method's parameters `a`, `b` and `c`, evaluates the three argument expressions `x+1`, `y` and `z[0]` to values v_1, v_2 and v_3, and then assigns the resulting values to the parameters, exactly as if assignments `a =` v_1 etc had been executed. That is, a parameter of primitive type is assigned a copy of the value of the argument, whereas a parameter of reference type is assigned a reference to the value of the argument.

Advanced Aspect. Thus Java always uses call-by-value, since a method parameter will either receive a copy of a primitive value or a copy of a memory reference to some object in the heap. By contrast, C# also has call-by-reference; see Appendix A.

```java
static int m(int a, int[] b, int c) { b[1] = a + 2 + c; return b[0]; }
int x = 10;
int[] y = new int[] { 21, 22 };
int[] z = new int[] { 30 };
int r = m(x+1, y, z[0]);
// Now y[1] is (10+1) + 2 + 30 = 43, and r is 21
```

Running the above Java code produces the following stack and heap contents:



3.16 Recursive method calls

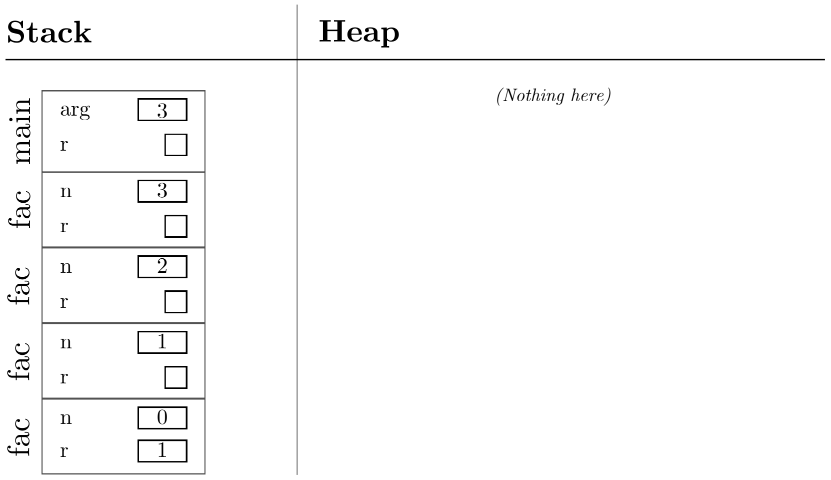
A method that may directly or indirectly call itself is recursive. When a method calls itself there will be multiple frames on the stack, each corresponding to a different call to the method.

The static method `int fac(int n)` below computes the factorial $n! = 1 \cdot 2 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$ of its argument `n`. The base case is $n = 0$, for which the result is 1. The inductive case, where $n \neq 0$, is computed as $(n-1)! \cdot n$.

When the Java code is executed, the initial call `fac(3)` will lead to a recursive call `fac(2)`, which will lead to a recursive call `fac(1)`, which will lead to a recursive call `fac(0)`, which will return 1 without any further calls.

```java
static int fac(int n) {
  int r = n==0 ? 1 : fac(n-1) * n;
  return r;
}
int arg = 3;
int res = fac(arg);
```

Running the above Java code produces this stack and heap contents, where we show the stack right before the last call to `fac` returns 1. At this point there are five frames on the stack, one for `main` and four for `fac`, corresponding to `n` being 3, 2, 1 and 0:

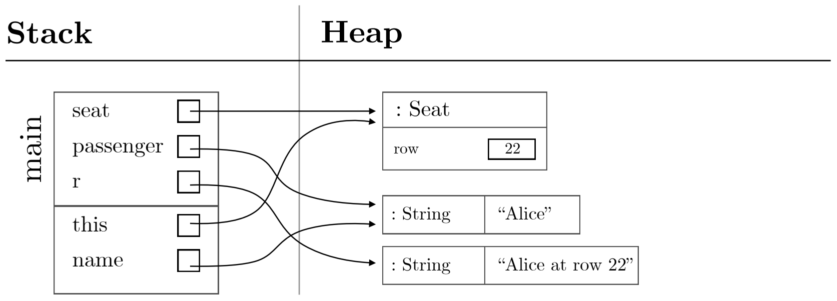


3.17 Instance methods and the this reference

A call `seat.card(p)` to an instance method `card(String name)` creates a new stack frame for holding the method's parameter `name`, evaluates the argument expression, and assigns the resulting values to the parameters, using call-by-value exactly as for a static method (Section 3.15). In addition, an implicit parameter `this` points to the object on which the instance method was called, here the object referred to by `seat`. This object is often called the receiver of the method call.

```java
class Seat {
  int row;
  public Seat(int row) { this.row = row; }
  String card(String name) { return name + " at " + row; }
}
Seat seat = new Seat(22);
String passenger = "Alice";
String r = seat.card(passenger);
```

Running the above Java code produces the following stack and heap contents:



3.18 Inner classes and multiple `this` references

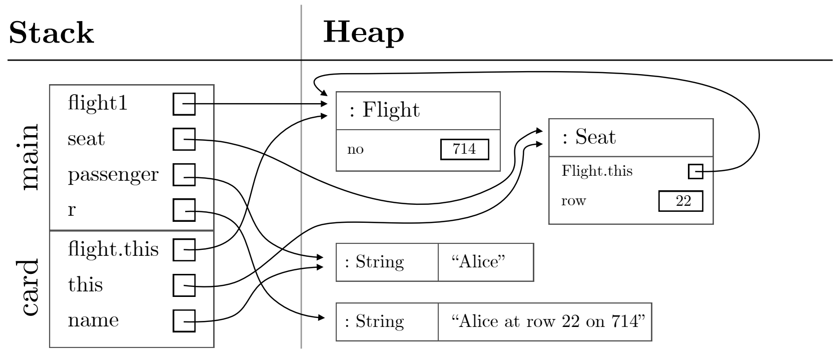
If class Seat is declared inside another class Flight and is not static, then it is an *inner class*, and every object instance of Seat will contain a reference to some instance of the enclosing class Flight. An instance method **`card`** in Seat can refer to fields of both the Seat object and the Flight object. For this to work, the stack frame for **`card`** will contain two **`this`** references, one for the Seat instance, called **`this`**, and one for the Flight instance, called **`Flight.this`**.

If class Seat were declared static inside class Flight, it would not be an inner class but a *nested class*, and Seat instances would not contain a reference to a Flight instance.

Advanced Aspect. The C# language does not have inner classes, only nested classes; see Appendix A.

```
class Flight {
  int no;
  public Flight(int no} { this.no = no; }
  class Seat {
    int row;
    public Seat(int row) { this.row = row; }
      String card(String name) { return name + " at " + row + "
    on " + no; }
  }
}
Flight flight1 = new Flight(714);
Seat seat = flight1.new Seat(22);
String passenger = "Alice";
String r = seat.card(passenger);
```

Running the above Java code produces the following stack and heap contents:



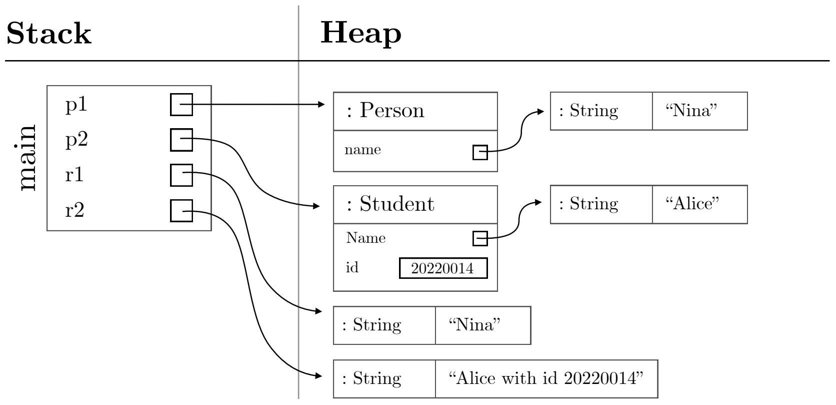
3.19 Virtual method calls and the run-time object type

In Java, every non-private instance method is *virtual*. This means that in a call `p2.show()` to the method, the run-time type of the value referred to by `p2`, that is the receiver object, determines which override of method `show` is called. A static method is non-virtual because it has no receiver object (`this`), and private method is non-virtual because it cannot be overridden.

Let us continue with example classes Person and Student from Section 3.6, but add an instance method `show` to Person and override it in subclass Student. We then call `p1.show()` and `p2.show()`, where both `p1` and `p2` have compile-time type Person, but the run-time types of the values they refer to differ (Person and Student) and that determines which `show` method is called in each case:

```java
class Person {
  ... as before
  String show() { return name; }
}
class Student extends Person {
  ... as before
  String show() { return name + " has id " + id; }
}
Person p1 = new Person("Nina");
Person p2 = new Student("Aline", 20220014);
String r1 = p1.show();         // Will call Person's show method
String r2 = p2.show();         // Will call Student's show method
```

Running the above Java code produces the following stack and heap contents:



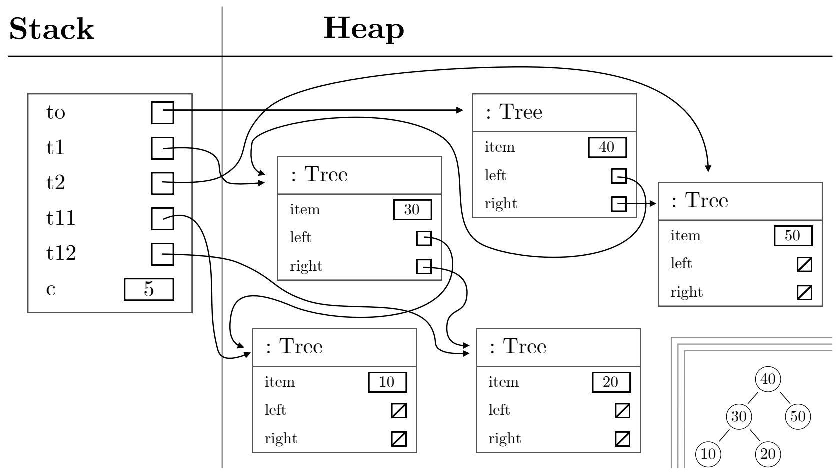
3.20 Representing a tree structure with objects

Objects are very useful for representing tree-structured data. Tree-structured data are found everywhere, in text documents (chapters, sections, paragraphs, sentences), JSON documents, XML documents, organization charts, abstract syntax of expressions and programs, and much more. Class Tree below can be used to represent trees where there is an integer in each tree node, and possibly a left and a right subtree; if a subtree field is `null`, then there is no subtree.

Method `count` computes the number of nodes in the tree; see Section 3.21. The code below builds builds the 5-node tree informally shown in the lower right corner of the figure at the bottom.

```java
class Tree {
  public int item;
  public Tree left, right;     // Tree node's subtrees
  public Tree(int item) { this.item = item; }
  public int count() {
    int r = 1 + (left==null ? 0 : left.count())
              + (right==null ? 0 : right.count());
    return r;
  }
}
Tree t0 = new Tree(40), t1 = new Tree(30), t2 = new Tree(50),
    t11 = new Tree(10), t12 = new Tree(20);
t0.left = t1; t0.right = t2; t1.left = t11; t2.right = t12;
```

Running the above Java code produces the following stack and heap contents:

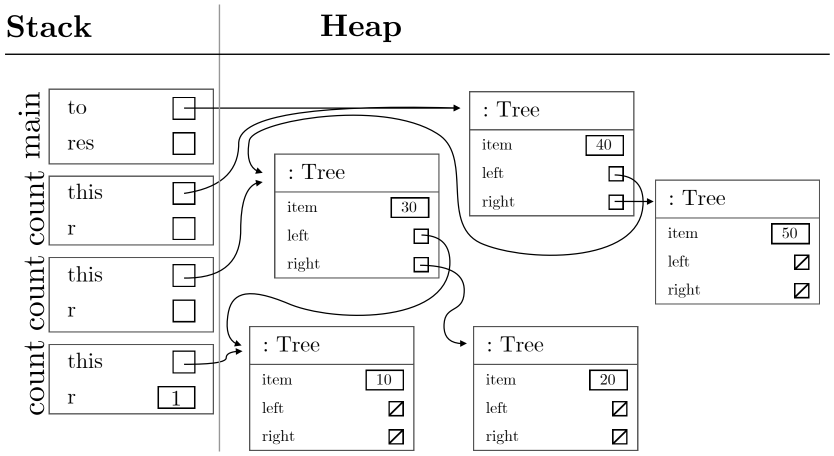


3.21 Recursive methods on tree structures

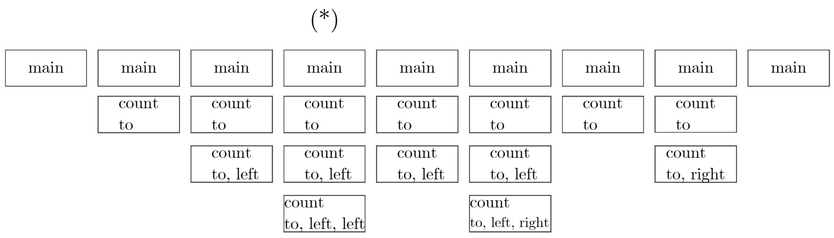
Method `count` from the example in Section 3.20 computes the number of nodes in a given tree. The method is recursive, so during the computation of `t0.count()` the stack will hold multiple frames corresponding to different calls to `count`.

```
res = t0.count();
```

Below we show the stack and head contents at the point where `main` has called `count` on `t0`, which has called `count` on `t0.left`, which has called `count` on `t0.left.left`, which has computed `r = 1` and is ready to return.



Later, there will be frames corresponding to calling `count` on `t0.left.right` and then `t0.right`; when all those calls are finished, `res` in the frame for `main` will contain 5, the number of Tree nodes. The total sequence of stacks is summarized below from left to right; the stack marked (*) is that shown above. The heap is unchanged throughout.



3.22 Throwing and catching exceptions

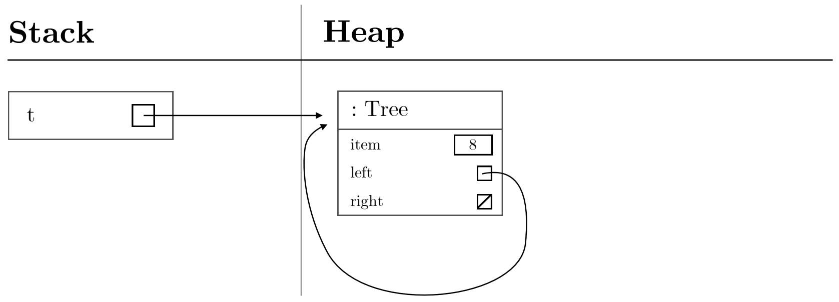
3.23 Cyclic references in the heap

The heap may contain reference cycles. This can be useful and is often unproblematic; Java certainly has no problem handling such cycles.

This is a continuation of the example in Section 3.20, using the same class Tree, but now intentionally creating a cyclic structure:

```java
Tree t = new Tree(8);
t.left = t;
```

Running the above Java code produces the following stack and heap contents:



The Tree object pointed to by `t` has itself as a left child, since `t.left == t`. One may think of this data structure as representing an infinite tree, or one might think that the data structure is ill-formed. Certainly, an attempt to compute `t.count()` would not terminate with a result; see Section 3.24.

Section 3.25 described how the garbage collector will remove heap objects that are not live, that is, not reachable from the stack. Since variable `t` refers to the Tree object above, it is live and hence will not be removed by the garbage collector. But if we make an assignment such as `t = new Tree(0)` or `t = null` so that there is no longer anything in the stack that refers to the old Tree object, it will be dead and can be garbage collected — despite there being a reference to that Tree object (from itself). Java's garbage collector handles cyclic references without problems, as does C#'s.

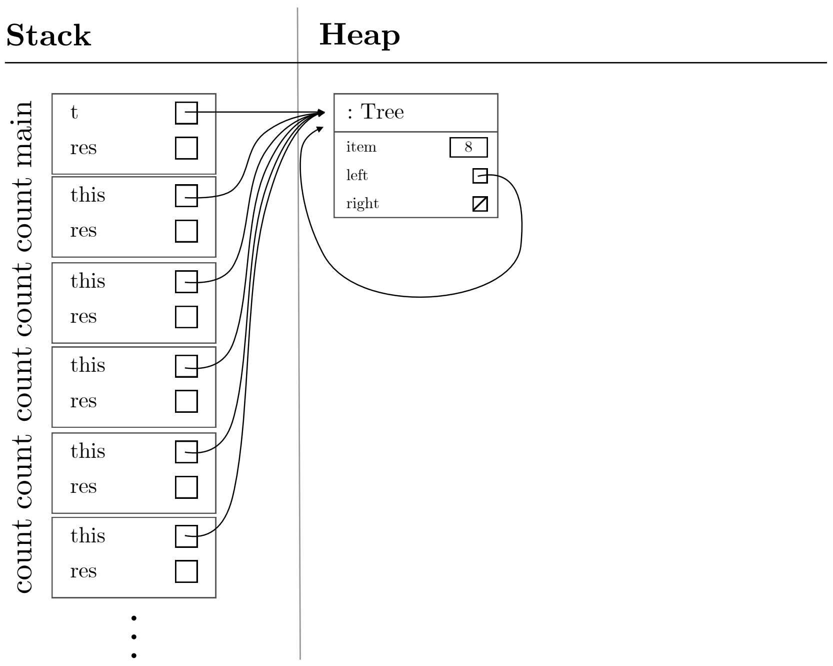
3.24 Infinite recursion on cyclic data

One reason to avoid cyclic data (Section 3.23) is that they may cause otherwise sensible and well-defined recursive functions such as `count` from Section 3.20 to go into an infinite recursion and cause a StackOverflowError exception at run-time.

This will happen if we attempt to compute `t.count()` with `t` as defined in Section 3.23:

```java
Tree t = new Tree(8);
t.left = t;
res = t.count();
```

Running the above Java code will attempt to create a stack containing an infinite number of frames, corresponding to infinitely many recursive calls to `count`. This is because each call to `count` on `t` will find that `t.left` is not `null`, and will try to recursively compute `t.left.count()`, but `t.left` equals `t`, so this would go on forever. In practice, the stack has no space for more stack frames, and the program execution will fail with a StackOverflowError:



3.25 Garbage collection of unreachable objects

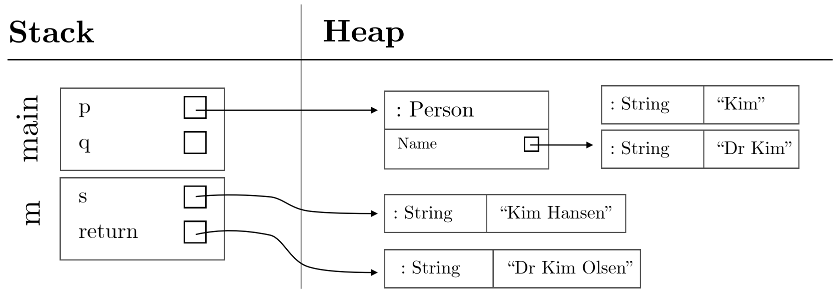
An object in the heap is *live* if there is a chain of references that leads from a stack frame to the object, possibly through other objects in the heap. An object that is live can be reached and used by the running Java program and therefore needs to remain in the heap. An object that is not live can be removed by the garbage collector, so that the heap memory previously occupied by the object can be reused for new objects.

The example in Section 3.5 showed how the string `"Junjun"` was allocated (and was initially live because reachable via `phd2.name`) but is not longer live after the assignment of a new value to `phd2.name`. Therefore that string object can be removed by the garbage collector; this will typically happen with some delay, when the Java system needs space for new objects. Sections 4.1 and 4.3 contain further examples of garbage collection.

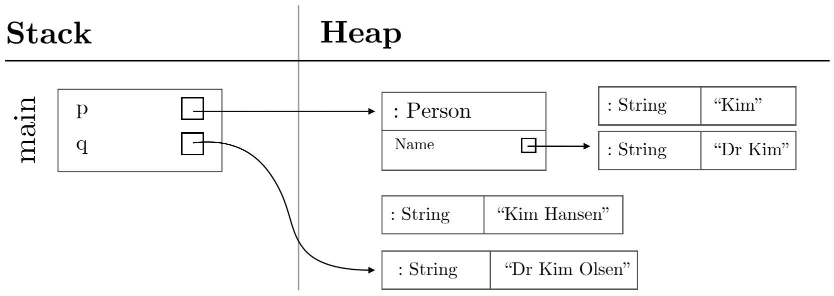
This example uses class Person from Section 3.5:

```java
static String m(Person p) {
  String s = p.name + " Hansen";
  p.name = "Dr " + p.name;
  return p.name + " Olsen";
}
Person p = new Person("Kim");
String q = m(p);
```

Method `m(p)` creates three new string objects, all live at the point right before the call `m(p)` returns, but the string formerly pointed to by `p.name` is not:



After the call has returned and `q = m(p)` has been performed, two of the new strings are live, reachable as `p.name` and `q.name`, but since variable `s` has been removed, the string it pointed to is no longer live and can be removed by the garbage collector:


Stack
Heap
main
p
q
: Person
Name
: String
"Kim"
: String
"Dr Kim"
: String
"Kim Hansen"
: String
"Dr Kim Olsen"

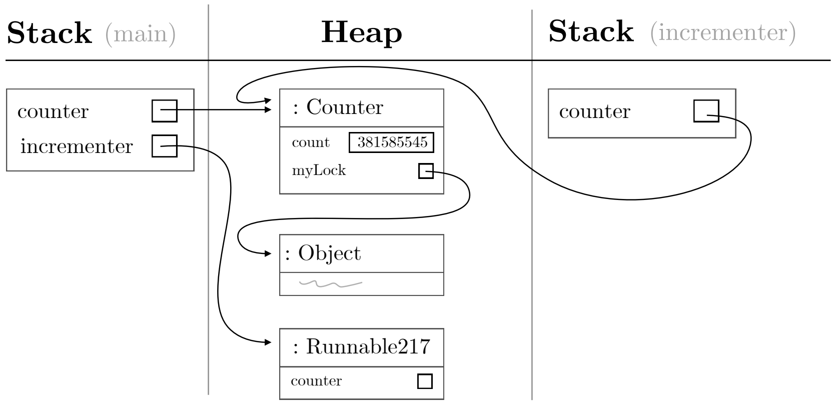
3.26 Multiple threads and multiple stacks

A thread is a sequential activity that may run concurrently with (that is, at the same time as) other threads. Each thread has its own stack for method calls, but all threads share the same heap. When a Java program starts, there is a single thread, the one that calls the `main` method.

The code below declares a class Counter for holding a long integer counter, it creates an instance of the counter, and creates a new thread from the lambda bound to the `incrementer` lambda. The closure for this lambda will hold a reference to the Counter object (as described in Section 3.14). When the incrementer thread is started, a stack (shown on the right) is created for that thread, with a stack frame that holds a reference to the Counter object in the shared heap (shown in the middle).

```
class Counter {
  private long count = 0;
  private final Object myLock = new Object();
  public void inc() { synchronized (myLock) { count = count + 1;
     } }
  public long get() { synchronized (myLock) { return count; } }
}
Counter counter = new Counter();
Runnable incrementer = () -> { while (true) counter.inc(); };
new Thread(incrementer).start();
while (true) { System.out.println(counter.get()); Thread.sleep
    (500); }
```

Running the above Java code produces the following stack and heap contents:



The Counter class has only private fields and all its public methods take the same lock `myLock` (using the `synchronized` statement) before manipulating the mutable state, that is, the `count` field; the class is a properly written monitor and is thread-safe. The thread-safety is easily undermined, though. If the next developer adds a public method to Counter, but forgets to take the lock, or declares the method synchronized so that it takes a different lock (namely the Counter object itself rather than `myLock`), then all thread-safety is lost.

3.27 Java 5 generics and erasure semantics

Java 5 introduced generic types and methods in 2004. They are implemented using so-called erasure semantics or homogenous representation at run-time, meaning that at run-time, the type parameter T in a generic type C<T> is replaced by Object or by a bound given on the type parameter. As a consequence there are some inefficiencies and limitations on Java generics. This is in contrast to C#, whose implementation of generic types and methods is more advanced and avoids these problems; see Appendix A.

[JP §21.10, §21.11]

4 Performance of Java programs

4.1 Time consumption

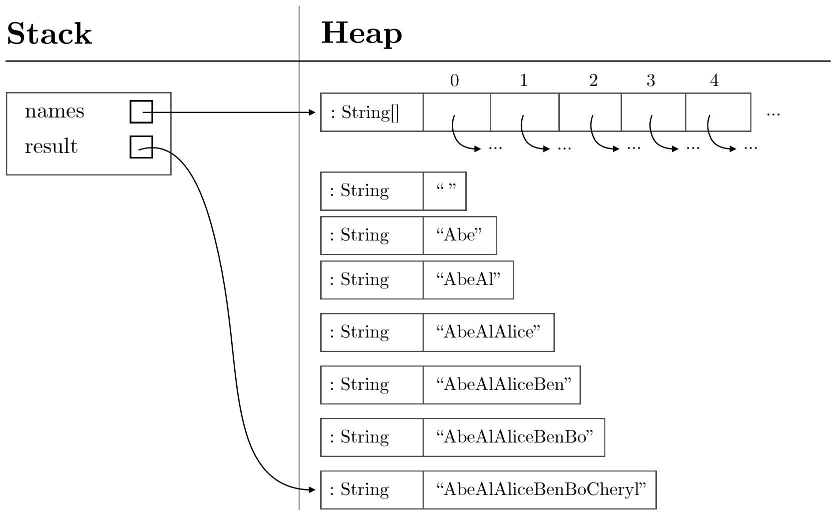
As can be seen from Sections 3.1, 3.2, 3.3 and 3.5, an assignment `x = y` of a variable `y` to another variable `x` just copies a primitive value, or copies a reference to a string, array or object (unless it also performs a primitive conversion or a boxing). In all cases, the assignment itself is very fast, copying only 4–8 bytes of memory.

Also, computing an assignment `x = e` where the right-hand side expression `e` involves only primitive values will usually be fast, unless it involves advanced mathematical functions such as `exp` or `sin`.

However, computing an assignment `x = e` where the right-hand side expression `e` involves heap-allocated objects may take a lot more time than one thinks. A typical mistake is to build a string by repeated concatenation `result += name`, which of course means `result = result + name`:

```
String[] names = { "Abe", "Al", "Alice", "Ben", "Bo", "Cheryl",
    ... };
String result = "";
for (String name : names)
  result = result + name;      // BAD repeated string
    concatenation
```

The innocent-looking string operator "`+`" (or "`+=`") here creates a new string object in each iteration, and each one is longer than the previous one. Running the above Java code produces this stack and heap contents, where one can see that a lot of intermediate string objects have been created:



The intermediate string objects are unreachable, so the garbage collector can remove them and the memory can be reused. Nevertheless a lot of time has been spent allocating and computing them.

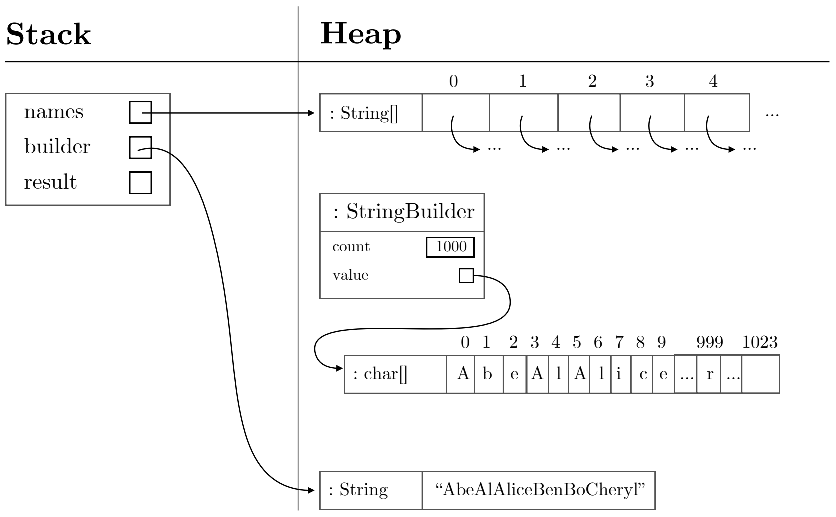
4.2 Memory consumption: allocation

A running program may inadvertently allocate a lot of short-lived objects in the heap, typically by boxing (Section 3.13) or by computing many intermediate objects, such as strings (Section 4.1). The many object allocations will take time, but if the objects are short-lived they will quickly be deallocated and the memory reused, so the heap will stay small.

Nevertheless, it is obviously best to avoid allocating a lot of superfluous objects, so the flawed string concatenation code from Section 4.1 should be replaced by this, using a single StringBuilder instead of allocating thousands of the short-lived strings:

```java
String[] names = { "Abe", "Al", "Alice", "Ben", "Bo", "Cheryl",
    ... };
StringBuilder builder = new StringBuilder();
for (String name : names)
  builder.append(name);   // GOOD string concatenation
String result = builder.toString();
```

Running the above Java code produces the following stack and heap contents:



4.3 Memory consumption: retention

Needlessly creating many short-lived objects in the heap is bad, because it makes the computation slower. But it is much worse to needless keep objects alive (reachable from the stack), because the heap may grow very large and the computation may fail with OutOfMemoryError.

Example: A queue can be represented by a linked list with `first` (dequeue) and `last` (enqueue) references, and `first` pointing to a dummy element:

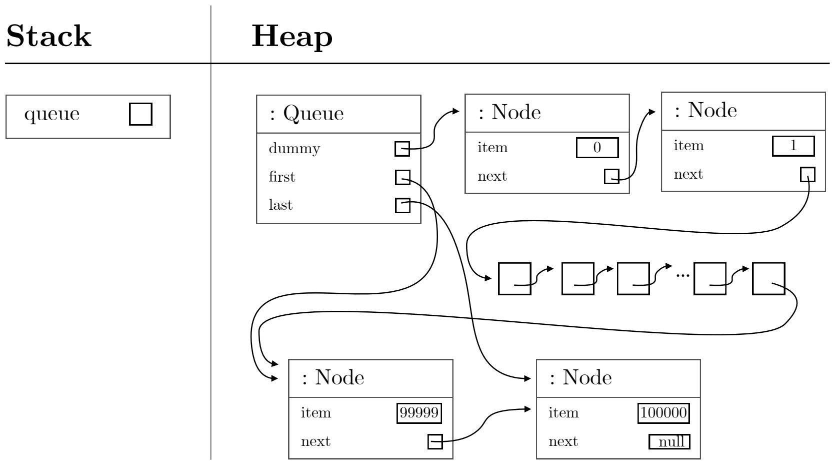
```
class Node { int item; Node next; }
class Queue {
  Node first, last; // Dequeue from first.next.item; empty if
    first==last
  public Queue { first = last = new Node(); } // Initial dummy
    node
  public void enqueue(int item) { Node n = new Node(); n.item =
    item; last.next = n; last = n; }

  public int dequeue() {
    if (first==last) throw new RuntimeError("Empty queue");
    else { first = first.next; return first.item; }
  }
}
```

This works fine: after 100,000 enqueues and 99,999 dequeues, there will be two live Node objects in the heap, reachable from `first`. One might instead, by mistake, bind the dummy node to a field:

```
Node dummy = new Node();
public Queue { first = last = dummy; } // Initialize with a
    dummy node
```

This would be *very bad*, because the field will keep the dummy Node object alive, and it will keep all the other 100,000 Node objects alive:



4.4 Time consumption of multi-threaded programs

Some sources of time waste and non-scalable parallelism, where throughput (the amount of work that gets done) does not grow linearly in the number of available compute cores:

- Lock contention: Multiple threads are ready to run (on separate cores) but are waiting to take the same lock, and then hold it for a long time, keeping other threads waiting. Symptom: Some cores are idle, CPU utilization is low. Mitigation: Reduce the time lock is held, possibly by reducing the use of shared mutable state. Use lock striping. Switch to lock-less optimistic concurrency.

- Busy wait: Multiple threads repeatedly test whether a condition has become true (because some other thread changed some data). Symptoms: CPU utilization is high, number of instructions executed is high, but little work gets done. Almost always a very bad idea. Mitigation: Use locks, barriers, semaphores, asynchronous computations, streams or events, which allow the threads to "wait" without consuming compute resources.

- Lock-less, optimistic concurrency, using compare-and-set: Multiple threads perform updates, but undermine each others' work by updating the state from which the optimistic computation started. Symptom: CPU utilization is high, number of instructions executed is high, throughput is low. Mitigation: Reduce the use of shared mutable state.

- Low-level cache effect: Different cores (running different threads) invalidate each others' memory caches, causing cache lines to cycle between the M and I states rather than stay in the E, M or S states (MESI protocol). Subtle point: Taking a lock very frequently can cause this to happen, since taking a lock requires M access of a memory location. Symptoms: CPU utilization is high, number of instructions executed is low. Mitigation: Reduce the use of shared mutable data.

- Subtle low-level cache effect: Mutable data values that "have nothing to do with each other" happen to be on the same cache line (each typically 64 bytes), and therefore unrelated updates can invalidate cache lines as per the previous item. Since locks are objects, this may even affect the taking of distinct locks, if they happen to be allocated on the same cache line. Symptoms as above. Mitigation: Rearrange field order in objects. Use memory "padding" to make sure that frequently updated object fields, array elements and heap-allocated objects (such as locks) are on distinct cache lines.

In languages with garbage collection, such as Java and C#, one way to "reduce the use of mutable state", as suggested above, is to use immutable data, causing more allocation and garbage collection, with little loss of efficiency.

5 Literature

CA Peter Sestoft: Computer arithmetics. Lecture slides, 2021; at https://www.itu.dk/people/sestoft/papers/computer-numbers-2021.pdf

DG David Goldberg: What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys, Volume 23 Issue 1, March 1991, pp. 5-48.

JLS James Gosling et al.: The Java Language Specification, March 2022, Oracle; at https://docs.oracle.com/javase/specs/jls/se18/jls18.pdf

JP Peter Sestoft: Java Precisely, 3rd edition 2016, MIT Press.

A C# run-time data storage

The C# programming language is in most respects very similar to Java, and most of the description above applies to C# also. However, C# has additional aspects, described in this appendix:

- user-defined value types (structs)
- reference parameters in method calls
- lvalue capture in anonymous functions (lambdas)
- non-virtual instance method calls `o.m(...)`, where the compile-time type of the receiver expression `o`, not the run-time type of the value of `o`, determines which method is called
- a more advanced implementation of generic types and methods

A future version of this document may elaborate on these aspects.