Introduction to Part 2 & Recap of Part 1

Lecture 9 of TDA 540 (Objektorienterad Programmering)

Carlo A. Furia Alex Gerdes

Chalmers University of Technology – Gothenburg University

Fall 2017

Welcome to Part 2 of the course!

The organization of Part 2 is very similar to Part 1:

- 6 lectures
- 4 labs
- lectures and lecture slides in English, but everything else in Swedish
- we will sometimes do short quizzes in class using `kahoot.it`
  - the quizzes are anonymous and not graded:
    only to get an idea of what is clear and what not
  - hopefully they'll make lectures a bit more entertaining!
- all the organizational details remain as in Part 1

Welcome to Part 2 of the course!

Main topics in Part 2:

- review of Part 1 (today)
- object-oriented features of Java
  - classes, attributes, and methods
  - inheritance and polymorphism
  - abstraction and interfaces
- event-driven programming
- some useful standard libraries

Suggestions

- The Java language is pretty big
  - in class, we will focus on significant examples without always covering all possible cases
  - look up the official documentation, as well as resources such as `stackoverflow.com`
- Ask questions and try out code snippets that we show in class
- There is a bit of overlapping between Part 1 and Part 2, as in Part 2 we will revise and extend some topics
- Establishing basic terminology and concepts is an important goal of the course

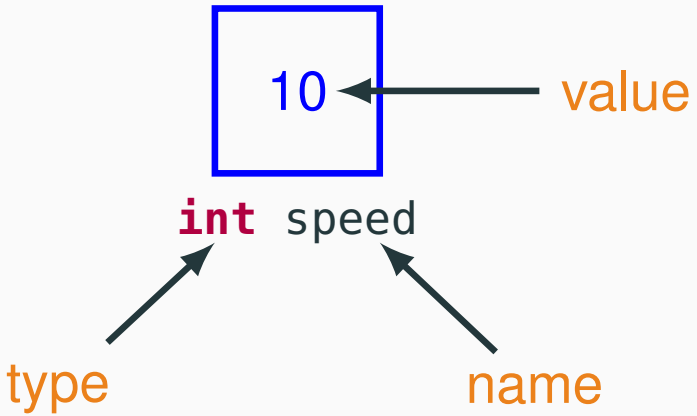
Pop quiz!

1. Go to `kahoot.it`
2. Enter PIN shown on projector screen
3. Pick a nickname and go!

Variables

Variables are a fundamental abstraction of data in programs.

- A variable represents a memory location storing a value that can be read and written to.
- A variable has a name (identifier), which provides a way to access the variable's content within a program's text.
- A variable has a type, which constraints what kind of data (possible values) the variable's memory location can contain.



Indentifiers

Variables, as well as other entities we define in a program such as user-defined classes, have names given by an identifier.

Java identifiers rules:

- identifiers are case sensitive: `speed` and `SPEED` are different identifiers
- the first character must be a letter, underscore `_`, or dollar sign `$`
- the following characters can be numbers, letters, underscore `_`, or dollar sign `$`

A variable's life

A variable:

- Must be declared before being used.
- May be initialized upon being declared.
- Its value can be read in an expression.
- Its value can be modified in an assignment.

OPERATION	CODE EXAMPLE	
declaration	`int speed;`	reserve room in memory for a variable with name `speed` and type `int`
initialization	`int speed = 10;`	set to `10` the initial value of `speed`
read/access	`if (speed > 5)`	use the current value of `speed` in an expression
write/modify	`speed = 8;`	change to `8` the value of `speed`

Declarations and Initialization

Variables must be declared before being used.
A variable declaration provides:

- the variable's type
- the variable's name (identifier)
- optionally, the initial value (initialization)

```
// declare a variable w of type int
int w;
// declare a variable x of type int, initialize x to 3
int x = 3;
```

- even if we do not initialize a variable when declaring it, we have to initialize it before first accessing its value.

We can declare together multiple variables with the same type:

```
// declare variables y and z, both of type int, initialize z to 4
int y, z = 4;
```

Types

Java is a (strongly) typed language. This means that every variable has a type associated with it.

A type constraints:

1. The (kinds of) values that a variable can take.
2. The operations that can be performed on variables of that type.

Example: a variable `speed` of type **`int`**:

1. `speed` can take any integer value between $-2^{31} =$ `Integer.MIN_VALUE` and $2^{31} - 1 =$ `Integer.MAX_VALUE`; and cannot take any other value (for example, 2^{40}, 0.33, $1/7$, $\sqrt{2}$, `"hello!"` are all forbidden values for a variable of type **`int`**).
2. we can perform arithmetic operations (`+`, `-`, `*`, `/`, `%`, ...), assignments, and comparisons with variables of other compatible numeric types.

Primitive types in Java

All Java types are partitioned into primitive and reference types.

Primitive types:

- 4 integer types of different size (`byte`, `short`, `int`, `long`)
- 2 floating point types of different size (`float`, `double`)
- 1 character type (`char`)
- 1 Boolean type (`boolean`)

About primitive types:

- Primitive types have names in lowercase letters
- We cannot define new primitive types: these 8 are all the primitive types that are available in Java
- In the first part of the course, we have mainly used primitive types

Reference types in Java

Reference types:

- 8 wrapper types (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`), each corresponding to a primitive type with the same (or similar) name
- the `String` type, representing strings of characters
- the `Array` type, representing sequences of values of homogeneous type, accessible by index
- many other types in the Java standard libraries

About reference types:

- Reference types have names that start with an uppercase letter
- Each reference type corresponds to a class with the same name
- We can define new reference types: this is what object-oriented programming is about!
- In the second part of the course, we will learn many new things about reference types and use them extensively

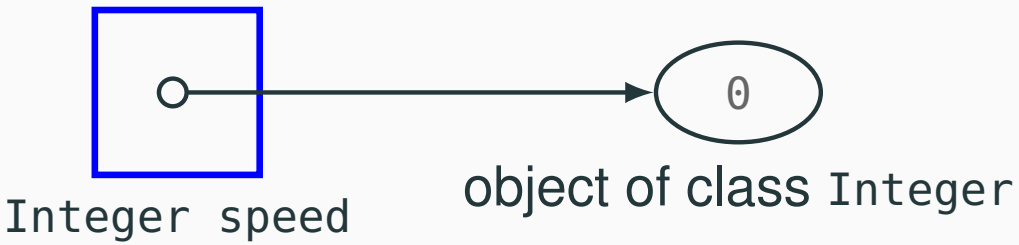
Initialization of reference types

We use the **new** keyword to initialize variables of reference type.

```
Integer speed = new Integer(0); // Initialize speed to 0
```

The expression `new Integer(0)` invokes the constructor of class `Integer`, which creates a new object of type `Integer`, initializes its stored value to `0`, and attaches the reference variable `speed` to the object. After the initialization:

`speed` points to an object of type `Integer`



We can initialize variables of a few "special" reference types (wrapper types, `Array`, `String`) without using **new**: see later for examples.

Type conversions

It is often necessary to combine values of different types.
In particular, for numeric types:

- Widening conversions: implicit, with no precision loss: from a smaller to a larger memory space
  - `byte` ⟶ `short` ⟶ `int` ⟶ `long`
  - `float` ⟶ `double`
  - `char` ⟶ `int` ⟶ `double`
- Narrowing conversions: explicit with a cast, with possible precision loss:
  - other combinations of numeric types
  - for example `long` to `int`
- Other conversions: implicit, with possible precision loss: conversion to floating point encoding, with varying precision
  - `int` ⟶ `float`
  - `long` ⟶ `float`
  - `long` ⟶ `double`

Type conversions: Examples

Widening conversions are implicit: we can use a value of the "smaller" type wherever a value of the "larger" type is needed.

- no precision loss example: **int** $\longrightarrow$ **long**

```
long companyValue = 651_500_000_000L;  // USD 651.5 billion
int companyTaxes = 7_682_000_000;  // USD 7.682 billion
   // companyTaxes implicitly converted to long:
long valueAfterTaxes = companyValue - companyTaxes;
```

Narrowing conversions are explicit: when we use a value of the "larger" type where a "smaller" type is needed we need a cast.

- precision loss example: **double** $\longrightarrow$ **int**

```
double width = 10.8;      int height = 11;
   // casting double to int, with precision loss:
   // 10.8 gets truncated to 10
int area = height * (int) width;
   // area is 110
```

Type conversions: Boxing

(Auto) boxing is the implicit conversion of a value of a primitive type to the corresponding wrapper reference type.

```
Integer balance = new Integer(0);
int interest = 120;
balance = balance + interest;     // boxing int to Integer
```

The rules of implicit type conversions for primitive types apply to the corresponding wrapper types.

```
Long companyValue = new Long(651_500_000_000L);
Integer companyTaxes = new Integer(2_140_000_000);
   // companyTaxes implicitly converted to Long:
Long valueAfterTaxes = companyValue - companyTaxes;
```

Initialization with boxing for wrapper types:

```
Integer balance = 100; // boxing to Integer object with value 100
```

Type conversions: Unboxing

(Auto) unboxing is the implicit conversion of a wrapper reference type to a value of the corresponding primitive type.

```
Integer balance = new Integer(0);
int balance_2 = balance;  // unboxing Integer to int
```

Expressions

An expression is obtained by combining variables and method calls with operators; it evaluates to a single value.

- Expressions must appear as part of statements – for example assignment statements

```
// expression without statement: error!
speed;
// expression as part of an assignment statement: OK
new_speed = speed;
```

- The simplest kinds of expressions are constants and variable references

```
speed = 3;  // constant expression with value 3
new_speed = speed; // expression 'speed'
```

- Expressions, like variables, have a type.
  The usual type compatibility rules apply.

Expressions

We build more complex expressions by combining simpler expressions using operators.

- Arithmetic expressions – numeric types:

```
speed + 3
2 * time
velocity / time
time % 60   // reminder of integer division: time / 60
```

- Comparison expressions – Boolean type:

```
initialSpeed < finalSpeed
3 == time  // equality
answer != 42 // non-equality
```

Expressions

We build more complex expressions by combining simpler expressions using operators.

- Boolean expressions – Boolean type:

```
true && false        // and (conjunction)
found || outOfBound  // or (disjunction)
!(speed < 0)         // not (negation/complement)
```

- Operator precedence and parentheses:

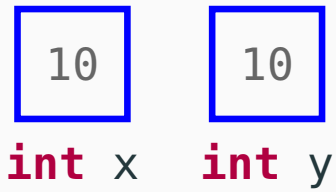
```
2 * (5 + 5) != 2 * 5 + 5  // value of the whole expression?
2 * 5 + 5 == 15           // value of the whole expression?
2 * (5 + 5) == 20         // value of the whole expression?
```

Equality comparison

The difference between primitive and reference types affects how the equality operators behave.

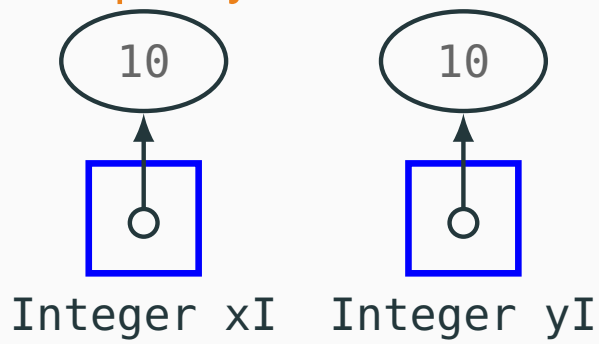
- For primitive types, == denotes value equality:

```
int x, y;
x = 10;
y = 10;
x == y // evaluates to true
```



- For reference types, == denotes reference equality:

```
Integer xI, yI;
xI = new Integer(10);
yI = new Integer(10);
xI == yI  // evaluates to false
```



- Method `equals` represents value equality for reference types:

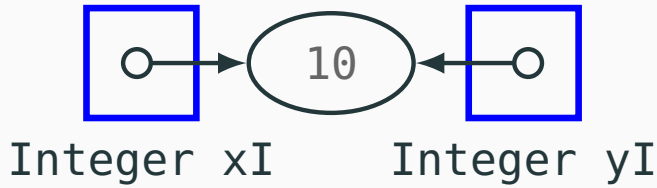
```
xI.equals(yI)  // evaluates to true
```

Equality comparison

The behavior of reference equality is tricky for “special” reference types (wrapper types and `String`) if initialized without using **`new`**.

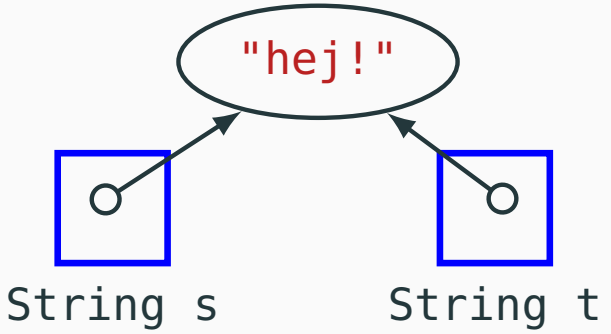
- Boxing constants to wrapper types gives one shared object per constant value.

```
Integer xI, yI;
xI = 10;
yI = 10;
xI == yI // evaluates to true: the objects are shared
```



- Constants of type `String` give one shared object per string value.

```
String s, t;
s = "hej!";
t = "hej!";
s == t // evaluates to true
```



Strings

Strings are sequences of characters. `String` is the Java class for strings, but strings have a "special" syntax:

- `s.length()` returns the length of a string `s`
- the empty string `""` has length zero, and is different from an uninitialized string variable
- operator `+` denotes string concatenation
- constant strings are shared objects; hence `==` is value equality (like method `equals`) even if `String` is a reference type

```
String s, t;              // declare strings 's' and 't'
s = "";                   // initialize 's' to empty string
t = "he" + "j!";          // concatenation with +
s = "hej!";               // s == t evaluates to true
s = new String("hej!"); // s != t but s.equals(t)
```

Arrays

Arrays are data structures to store sequences of elements of the same type. `Array` is the Java class implementing arrays, but arrays have a "special" syntax: for an array variable `a`:

- `a.length` denotes `a`'s fixed length (number of elements)
- `a`'s elements are stored at integer indexes from `0` to `a.length - 1` (inclusive)
- initialized arrays store a default value in their slots

```
int[] a;                 // declare array 'a' of int
a = new int[5];          // initialize 'a' to 5 elements
a[0] = 100;              // store 100 in position 0
a[a.length - 1] = 8;     // store 8 in last position
int[] b = new int[4];    // initialize 'b' to 4 elements
a[0] = b[0];             // b[0] is the default int value 0
int[] c = {1,1,1,1,2}; // initialize whole array at once
```

Default values

TYPE	DEFAULT VALUE
value integer types (**byte**, **short**, **int**, **long**)	`0`
value floating point types (**float**, **double**)	`0.0`
value character type (**char**)	`\u 0000`
value Boolean type (**boolean**)	**false**
reference types	**null**

Default values do not apply to local variables, but only to array content and attributes (see later).

Statements

A statement is a complete instruction that can be executed.

- Declarations are the simplest kind of statement, which only has an implicit effect when executed.
- Assignments change the value stored in variables:

```
speed = distance / time;
balance = 11_000;
```

- Method calls (also called “method invocations”):

```
int[] a = new int[100];          // 100-element int array
java.util.Arrays.fill(a, 42);  // fill ‘a’ with the number 42
```

- Control flow statements determine the order in which statements are executed: conditionals and loops:

```
if (velocity > 0)
   speed = velocity;
else
   speed = -1 * velocity;
```

```
for (int i = 0; i < a.length; a++)
  total = total + a[i];
```

Assignments

Every assignment has the form:

```
variable = expression;
```

- `variable` (the target of the assignment) is a single variable name
- the type of *expression* must be compatible with the type of `variable`

Executing an assignment:

1. evaluate *expression* to determine its value *v*
2. update `variable`'s value to *v*

Side effects

An expression has side effects if evaluating the expression may change some variables' values implicitly.

- Self-increment and self-decrement operators

```
int balance = 0, interest = 10;
balance = interest++;  // 1. evaluate interest
                       // 2. assign its value to balance
                       // 3. increment interest
balance = --interest;  // 1. decrement interest
                       // 2. evaluate interest
                       // 3. assign its value to balance
```

- Method calls in expressions may also have side-effects (we will see examples later)

Blocks and scope

Blocks group together statements to create compound statements.

```
{ // outer block begins
  int x = 0, y = 1;
  { // inner block begins
    int z = 2;
    y = z + 1; // OK: y declared in outer block
  } // inner block ends
  y = z + 3; // Error: z declared in inner block, not available here
} // outer block ends
```

- Blocks are marked by curly braces `{ ... }`
- A block can appear wherever a single statement can go
- Blocks can be nested inside other blocks
- Variables declared inside a block are only visible within the block (this includes other blocks nested inside the block)
- The visibility of a variable is also called scope

Conditionals: if-then-else

Conditionals determine which statements are executed according to the value of an expression (the condition).

if-then-else conditional:

```
if (amount < balance)
   // then branch
   balance = balance - amount;
else
   // else branch
   System.out.println("Cannot withdraw amount!");
```

- The condition is a Boolean expression
- Thus, the then and else branches are mutually exclusive: exactly one of them executes
- The else branch is optional: it may be omitted
- The then and else branches can be single statements or a block of statements

Conditionals: switch

Switch case statement:

```java
switch (balance) {
  case 0:
    System.out.println("You're broke!");
    // fall-through behavior without a break!
  case 1_000_000:
    System.out.println("You're rich!");
    break;
  default: // if all other cases are false
    System.out.println("You're average!");
}
```

1. Go through the cases in order, until a case matches
2. If no case matches, go to the **`default`** (if it exists)
3. Execute from the matching point on until a **`break`** (if it exists)

- Every case must be a constant expression
- The variable of **`switch`** can only be of type **`byte`**, **`short`**, **`int`**, **`char`**, their wrapped counterparts, `String`, or an **`enum`** type (see later).

Loops

Loops repeat (iterate) the execution of statements until a condition (Boolean expression) becomes true.

While loop:

```
int sum = 0; i = 0;
while (i < a.length) {
    sum = sum + a[i];
    i++;
}
// sum of all values in array ‘a’
```

Do loop:

```
int sum = 0; i = 0;
do {
    sum = sum + a[i];
    i++;
} while (i < a.length);
// sum of all values in array ‘a’
// only works if ‘a’ is not empty
```

Loops

Loops repeat (iterate) the execution of statements until a condition (Boolean expression) becomes true.

For loop:

```
int sum = 0;
for (int i = 0; i < a.length; i++) {
   sum = sum + a[i];
}
// sum of all values in array ‘a’
```

For-each loop:

```
int sum = 0;
for (int v : a) {
   // v takes all values in array ‘a’,
   // one per iteration
   sum = sum + v;
}
// sum of all values in array ‘a’
```

The for-each loop, also called enhanced for, can only iterate over arrays or collections (see later).

Classes

We only had a glimpse of classes during the first part. In the second part, we will learn many more things about them.

```
class Interest {   // in a file Interest.java

  static double interestYear(int year)   // method declaration
  {  return (year - 2010) / 100.0;  }

  // entry point of program
  public static void main(String[] args) {
    double interest = 0;
    int oldYear = 2016;
    interest = interestYear(oldYear - 3); // method call
    System.out.println("The interest for " + year
                       + " is " + interest);
  }
// ...
```

Methods

Methods are one kind of class members. In the second part, we will also learn more about method declaration and usage.

```
public static double interestYear(int year)  // method signature
{  // method body: implementation/definition
   return (year - 2010) / 100.0;
}
```

- `public` defines the method's visibility
- `static` identifies a class method
- `double` is the return type
- `int year` is the argument (also called parameter) declaration

Arguments

A method signature declares the types of the return (output) and (input) arguments:

```
double interestYear(int year)
```

- the return value has type `double` and is assigned by a `return`
- the input argument is available as a local variable named `year` within the method's body
- `year` is called formal argument

A method call must match the types and order of the arguments:

```
interest = interestYear(oldYear - 3)
```

- arguments are identified by their position (in this example, there is only one argument)
- expression `oldYear - 3` must have type compatible with `int` (see signature)
- `oldYear - 3` is called actual argument

How method calls work

```
double interestYear(int year)
```

```
interest = interestYear(oldYear - 3)
```

Java method calls are by value/copy:

IN GENERAL	IN THE EXAMPLE
each actual argument is evaluated	evaluate `oldYear - 3` to `2013`
the corresponding formal argument is initialized to the value	initialize `year` to `2013`
the called method's body is executed	execute `interestYear`
when execution reaches a `return e` statement, expression `e` is evaluated	evaluate `e` to `0.03`
the value becomes the value of the method call expression in the caller	`interestYear(oldYear - 3)` evaluates to `0.03`
execution continues in the caller	variable `interest` is updated to `0.03`

How method calls work

```
void dontSet(int v) {
    v = 10;
}
```

```
void set(int[] a) {
    a[0] = 10;
}
```

Java method calls are by value/copy:

- changes to the formal argument in the called method's body do not affect the actual argument in the caller
- however, the called method can still change the value of objects attached to references (variables of reference types)

```
int x = 0;
dontSet(x);
// x is still 0
```

```
int[] z = {0, 0};
set(z);
// z[0] is 10
```

How method calls work: primitive type arguments

Java method calls are by value/copy: with arguments of primitive type, the only way the called method (callee) can send information to the caller is via `return`.

```
void dontSet(int v) {
  v = 10;
}
```

`int v`

```
int x = 0;
dontSet(x);
// x is still 0
```

`int x`

How method calls work: primitive type arguments

Java method calls are by value/copy: with arguments of primitive type, the only way the called method (callee) can send information to the caller is via `return`.

```
void dontSet(int v) {
  v = 10;
}
```

`int v`

```
int x = 0; ⟸
dontSet(x);
// x is still 0
```

`int x`

How method calls work: primitive type arguments

Java method calls are by value/copy: with arguments of primitive type, the only way the called method (callee) can send information to the caller is via `return`.

```
void dontSet(int v) {
  v = 10;
}
```

`int v`

```
int x = 0;
dontSet(x); ⟸
// x is still 0
```

`int x`

How method calls work: primitive type arguments

Java method calls are by value/copy: with arguments of primitive type, the only way the called method (callee) can send information to the caller is via `return`.

```
void dontSet(int v) {
  v = 10; ⟸
}
```

`int v`

```
int x = 0;
dontSet(x); ⟸
// x is still 0
```

`int x`

How method calls work: primitive type arguments

Java method calls are by value/copy: with arguments of primitive type, the only way the called method (callee) can send information to the caller is via `return`.

```
void dontSet(int v) {
  v = 10;
}
```

`int v`

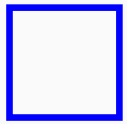
```
int x = 0;
dontSet(x);
// x is still 0 ⟸
```

`int x`

How method calls work: reference type arguments

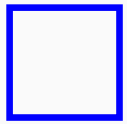
Java method calls are by value/copy: with arguments of reference type, the called method (callee) can send information to the caller also indirectly by modifying shared objects.

```
void set(int[] a) {
  a[0] = 10;
}
```



int[] a

```
int[] z = {0, 0};
set(z);
// z[0] is 10
```



int[] z

Note that the arguments are handled as for primitive types, but with reference types a reference is copied.

How method calls work: reference type arguments

Java method calls are by value/copy: with arguments of reference type, the called method (callee) can send information to the caller also indirectly by modifying shared objects.

```
void set(int[] a) {
  a[0] = 10;
}
```

```
int[] z = {0, 0}; ⟸
set(z);
// z[0] is 10
```

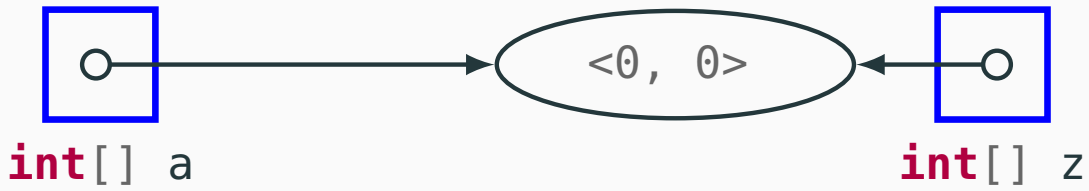
Note that the arguments are handled as for primitive types, but with reference types a reference is copied.

How method calls work: reference type arguments

Java method calls are by value/copy: with arguments of reference type, the called method (callee) can send information to the caller also indirectly by modifying shared objects.

```
void set(int[] a) {
  a[0] = 10;
}
```

```
int[] z = {0, 0};
set(z); ⟸
// z[0] is 10
```



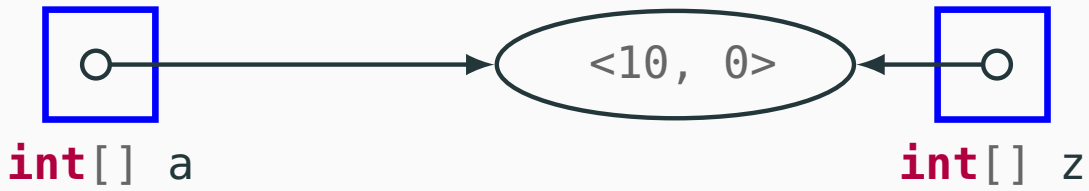
Note that the arguments are handled as for primitive types, but with reference types a reference is copied.

How method calls work: reference type arguments

Java method calls are by value/copy: with arguments of reference type, the called method (callee) can send information to the caller also indirectly by modifying shared objects.

```
void set(int[] a) {
  a[0] = 10; ⟸
}
```

```
int[] z = {0, 0};
set(z); ⟸
// z[0] is 10
```



Note that the arguments are handled as for primitive types, but with reference types a reference is copied.

How method calls work: reference type arguments

Java method calls are by value/copy: with arguments of reference type, the called method (callee) can send information to the caller also indirectly by modifying shared objects.

```
void set(int[] a) {
  a[0] = 10;
}
```

```
int[] z = {0, 0};
set(z);
// z[0] is 10 ⟸
```

Note that the arguments are handled as for primitive types, but with reference types a reference is copied.

Exceptions

Exceptions are objects used to signal unusual (often erroneous) conditions. Exception handling code specifies what the program should do when exceptions occur.

```
int n;  // What is the advantage of declaring ‘n’ outside try block?
Scanner sc = new Scanner(System.in);
try {
   n = sc.nextInt();   // may throw exception
   System.out.println("Found integer " + n);
} catch (InputMismatchException e) { // what to do when an exception
                                     // of given type is thrown
   System.out.println("Invalid integer as string!");
} finally {  // what to do after try/catch is executed
             // regardless of whether an exception was thrown
   sc.close();
}
```

Exceptions: `try` with resources

The try with resources mechanism introduces an implicit **`finally`** block. It is convenient when managing resources that must be opened and closed.

```
int n;
try (Scanner sc = new Scanner(System.in)) {
   n = sc.nextInt();   // may throw exception
   System.out.println("Found integer " + n);
} catch (InputMismatchException e) { // what to do when an exception
                                     // of given type is thrown
   System.out.println("Invalid integer as string!");
}  // sc.close() implicitly executed when
   // execution reaches this point
```