Introduction to Java programming

Data types, operators, flow control & methods

Michiel Noback
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Introduction

- This presentations has four parts

1. **data types**
2. **operators**
3. **flow control**
4. **methods**

PART 1: Data types

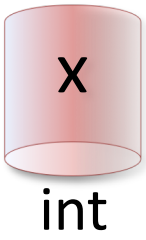
- Data forms the **state** of an object
- The “atoms” of data are the **primitive types**: booleans, integers, floating point numbers, characters and -not exactly a primitive type- strings
- Conversions between types is often possible and called **(type) casting**

Primitive types

Type	Default value	Range	example
boolean	false	NA	boolean toggle = true
char	\u0000	0 to 2^{16}-1 or \u0000 to \uFFFF	char guanine= 'G'
byte	0	-2^{7} to 2^{7}-1 or -128 to 127	byte b = 100;
short	0	-2^{15} to 2^{15}-1 or -32768 to 32767	short s = 100;
int	0	-2^{31} to 2^{31}-1 or -2147483648 to 2147483647	int i = 100;
long	0	-2^{63} to 2^{63}-1 or -9223372036854775808 to 9223372036854775807	long l = 100L;
float	0.0f	1.4E-45 to 3.4E+38 (plus or minus)	float f = 0.01f; f = 1e22f;
double	0.0	4.9E-324 to 1.8E+308 (plus or minus)	double d = 3.1415 d = 31415e-4;

Creating a primitive

```
int x = 7;
```



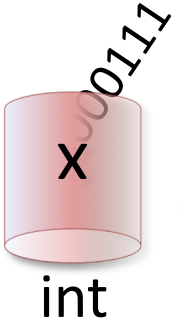
Declare the variable: space is reserved for the data type

```
int x = 7;
```

00000111

Create the variable: the bit pattern is constructed

```
int x = 7;
```



Assign the value: put the bit pattern into the reserved space

Primitive types in action

```
public class JavaTypesDemo {

    public static void main(String[] args) {
        int killCount = 42;          //INTEGER counts the number of kills
        boolean alive = false;       //BOOLEAN for yes/no variables; indicates alive status
        double killAverage = 10.55; //DOUBLE for floating point values; indicates the
                                     //average number of kills per life cycle
        char playMode = 'N';         //CHARACTER for single letter values; stores mode
                                     //of the game 'N'=No mercy 'S'= Sissy
        String player = "ZZZZZombie";//STRING for text values; the name of the player

        System.out.println( "player =       " + player );
        System.out.println( "alive =        " + alive );
        System.out.println( "play mode =    " + playMode );
        System.out.println( "kill count =   " + killCount );
        System.out.println( "kill average = " + killAverage );
    }

}
```

```
 $java JavaTypesDemo
player =       ZZZZZombie
alive =        false
play mode =    N
kill count =   42
kill average = 10.55
```

Declaring and initializing primitives

- Data variables have to be declared and initialized, but you can do them in one effort:

```
//Declare only
int killCount;
int lives;

//and initialize later
killCount = 0;
lives = 0;

//OR declare and initialize
int killCounter = 42;
int livesLived = 5;

//OR declare both in one go
int kills, livesNumber;
```

Using primitives

```
int x = 10;
int y = 20;

int squareSurface = y * y;
double circleSurface = Math.PI * (0.5 * y * 0.5 * y);
double square2circleRatio = squareSurface / circleSurface;
double div = x / y;

System.out.println( "squareSurface =       " + squareSurface );
System.out.println( "circleSurface =       " + circleSurface );
System.out.println( "square2circleRatio = " + square2circleRatio );
System.out.println( "div =                 " + div );
```

The Math class provides some nice constants

```
 $java JavaTypeOperationsDemo
squareSurface =       400
circleSurface =       314.1592653589793
square2circleRatio = 1.2732395447351628
div =                 0.0
```

10 / 20 = 0.0 ???

Converting between data types

- Sometimes you need to convert one type into another.
- The mechanism to do this is called **casting**

```
double div = x / y;
```

```
div =                    0.0
```

Clearly, 10 / 20 = 0.0 is not OK

```
double div = (double)x / y;
```

```
div =                    0.5
```

Perform a **cast** from **int** to **double** on variable x

(Type) casting

- Casting is converting the type of a variable
- Casting can only be safely done from lower to higher information content:

A **safe** cast from int to double. You do not need to do this explicitly

```
int x = 42;
double d = (double)x; //explicit
//double d = x; (implicit) is also
legal
double pi = 3.1415;
int y = (int)pi;
```

An **unsafe** cast from double to int. You need to do this explicitly

Using data types

- You can do all sorts of things with your variables

```
char letter = 'A';
System.out.println( "letter * x = " + (letter * x) );

String name = "ZZZZZomby";
System.out.println( "name + x =    " + (name + x) );
```

```
letter * x = 650
name + x =    ZZZZZomby10
```

Multiplying a char?

Adding to a String is *concatenating*

Limits to using primitives

- Some operations are illegal

```
char letter = 'A';
System.out.println( "letter * x =          " + (letter * x) );

String name = "ZZZZZomby";
System.out.println( "name + x =           " + (name + x) );
System.out.println( "name * x =           " + (name * x) );
```

Big fat compiler error!

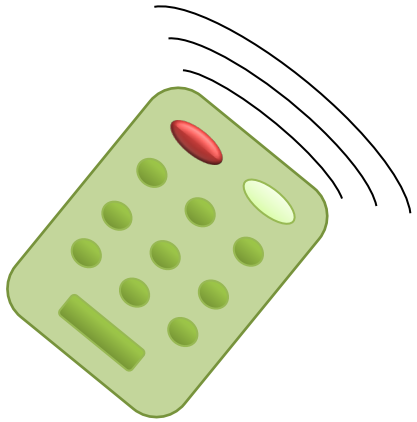
```
$javac JavaTypeOperationsDemo.java
JavaTypeOperationsDemo.java:23: operator * cannot be applied to java.lang.String,int
                System.out.println( "name * x =           " + (name * x) );
                                                               ^
1 error
```

Reference type variables

- In Java, reference types are everywhere; they point to the object instances
- Strings, arrays, and every object you will ever encounter are all reference types.

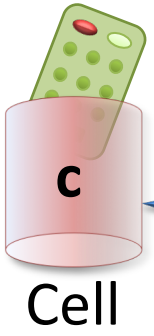
Primitives and references

- With primitive variables, the value of the variable is...*the value*.
- With reference variables, the value of the variable is *bits representing a way to get to a specific object* (like a remote control)



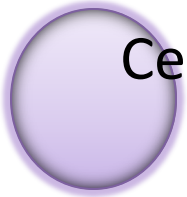
creating a reference type variable

```
Cell c = new Cell();
```



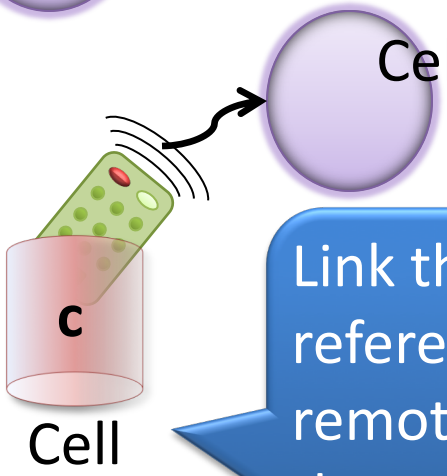
Declare the variable: space is reserved for a **reference** of type Cell

```
Cell c = new Cell();
```



Create the **object**

```
Cell c = new Cell();
```



Link the object and the reference: program the remote control to access the created object

Type String

- Some special attention to type String
- It is actually **NOT** a primitive type (you could have guessed from the name, which starts with a capital)
- It is a full-fledged Java class that can be instantiated and that has many nice methods
- It is however **immutable** as in most languages

Type String

```
String dnaOne = "AGAGGTCTAGCTGACTGAC";
String dnaTwo = "GGTCTAGCTG";
String dnaThree = "GGtctAGctg";
String dnaFour = dnaThree.toUpperCase();

System.out.println( "character at position 6: " + dnaOne.charAt( 5 ) );
System.out.println( "dnaOne contains dnaTwo: " + dnaOne.contains( dnaTwo ) );
System.out.println( "dnaTwo equals dnaThree: " + dnaTwo.equals( dnaThree ) );
System.out.println( "dnaTwo equals dnaThree ignoring case: " + dnaTwo.equalsIgnoreCase( dnaThree ) );
System.out.println( "dnaOne starts with \"AGAGGT\": " + dnaOne.startsWith( "AGAGGT" ) );
System.out.println( "dnaThree to uppercase: " + dnaFour );
System.out.println( "dnaOne to char array: " + dnaOne.toCharArray().toString() );
```

```
$ java StringDemo
character at position 6: T
dnaOne contains dnaTwo: true
dnaTwo equals dnaThree: false
dnaTwo equals dnaThree ignoring case: true
dnaOne starts with "AGAGGT": true
dnaThree to uppercase: GGTCTAGCTG
dnaOne to char array: [C@3e25a5
```

Here, we printed the **REFERENCE VALUE.**

Creating Strings

- String is a Java class, but is special since it has a unique way of creating instances

```
/*shortcut to instantiating a String object*/
String dnaOne = "AGAGGTCTAGCTGACTGAC";
/*official object instantiation*/
String dnaFive = new String( "AGAGGTCTAGCTGACTGAC" );
/*returns true*/
System.out.println( "dnaOne equals dnaFive: " + dnaOne.equals( dnaFive ) );
/*returns false*/
System.out.println( "dnaOne == dnaFive: " + (dnaOne == dnaFive) );
```

Comparing Strings

- String comparison is, like all java objects, **not** to be done using the equality test operator
- Use == for primitive types OR comparison of object references

```
/*shortcut to instantiating a String object*/
String dnaOne = "AGAGGTCTAGCTGACTGAC";
/*official object instantiation*/
String dnaFive = new String( "AGAGGTCTAGCTGACTGAC" );
/*returns true*/
System.out.println( "dnaOne equals dnaFive: " + dnaOne.equals( dnaFive ) );
/*returns false*/
System.out.println( "dnaOne == dnaFive: " + (dnaOne == dnaFive) );
```

```
$ java StringDemo
dnaOne equals dnaFive: true
dnaOne == dnaFive: false
```

Comparing object references or object state

- The equality operator (**==**) looks whether two object references are equal
- **equals()** is a method (of class String) that compares the contents/state of two objects (in the case of Strings, their character contents)

To be a primitive

```
public class ReferenceDemo1 {
    public static void main(String[] args) {
        ReferenceDemo1 demo = new ReferenceDemo1();
        demo.start();
    }

    public void start(){
        int x = 42;
        System.out.println( "variable x has value " + x );
        changeVariable( x );
        System.out.println( "variable x has value " + x );
    }

    public void changeVariable( int number ){
        System.out.println( "variable number has value " + number );
        number = 1024;
        System.out.println( "variable number has value " + number );
    }
```

Primitives are always COPIED when passed to a method

```
$ java ReferenceDemo1
variable x has value 42
variable number has value 42
variable number has value 1024
variable x has value 42
```

Or not to be a primitive

```
public void start(){
    Cell x = new Cell();
    x.size = 42;
    System.out.println( "cell size has value " + x.size );
    changeVariable( x );
    System.out.println( "cell size has value " + x.size );
}

public void changeVariable( Cell cell ){
    System.out.println( "guest cell has size " + cell.size );
    cell.size = 1024;
    System.out.println( "guest cell has size " + cell.size );
}
```

```
cell size has value 42
guest cell has size 42
guest cell has size 1024
cell size has value 1024
```

Objects are passed as REFERENCE! A copy is made of the reference value, not of the object itself!

String objects are immutable

```
public void start(){
    String x = "Hello, world";
    System.out.println( "variable x has value " + x );
    changeVariable( x );
    System.out.println( "variable x has value " + x );
}

public void changeVariable( String quote ){
    System.out.println( "variable quote has v
    quote = "Goodbye, World";
    System.out.println( "variable quote has v
}
```

Here, a new String object was created. The existing one was not changed but discarded.

It looks as if String behaves as a primitive!
That is NOT true!
Strings are IMMUTABLE

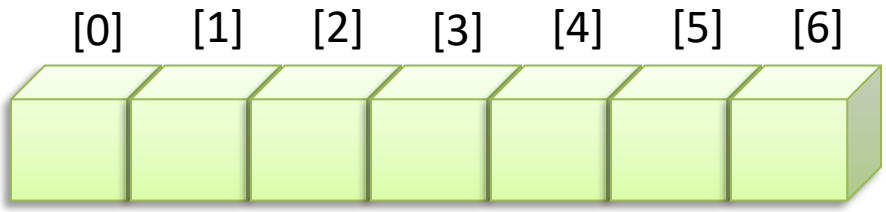
```
variable x has value Hello, world
variable quote has value Hello, world
variable quote has value Goodbye, World
variable x has value Hello, world
```

Arrays

- Just like any programming language, Java knows about arrays
- Arrays are ordered collections of variables *of the same type*
- These variables are not individually named and managed, but collectively
- In real life, you don't use arrays too much because other types (ArrayList) are more convenient

Arrays

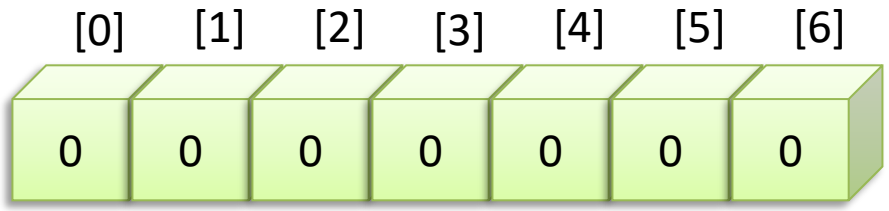
- To get to an element of an array, you use an **index**
- Java array indexing starts at 0



Arrays

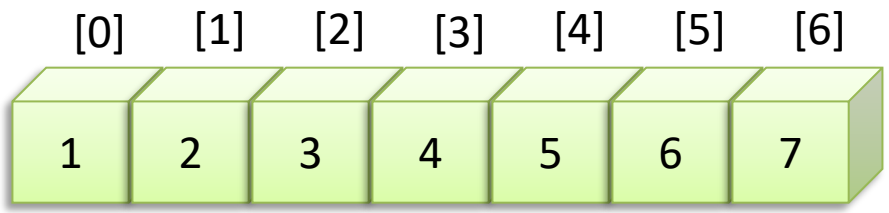
- To create an array, you have to say how many elements it will hold, in one of two ways:

```
int[] numbers = new int[7];
```



In Java, arrays are filled with default values when created

```
int[] numbers = {1,2,3,4,5,6,7};
```

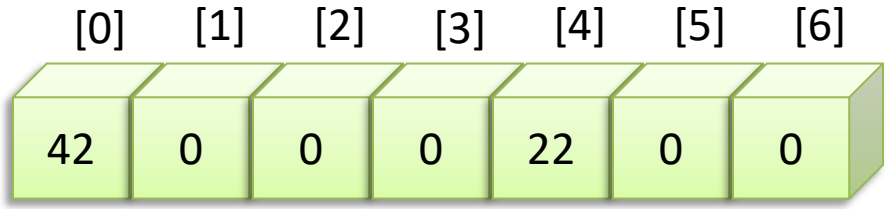


An array literal

Arrays

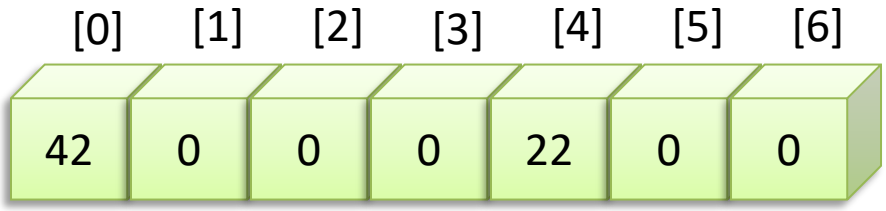
- To change elements of an array, you will have to use the index of the element you want to access:

```
int[] numbers = new int[7];
numbers[0] = 42;
numbers[4] = 22;
```

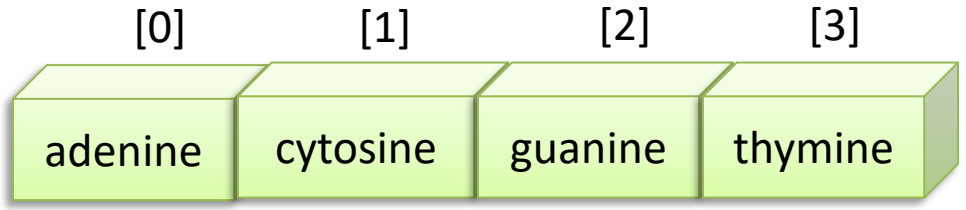


Arrays

- You can have arrays of primitives, such as ints, booleans etc...



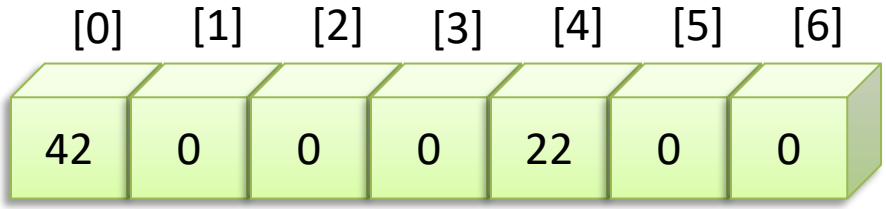
- but also of objects such as Strings



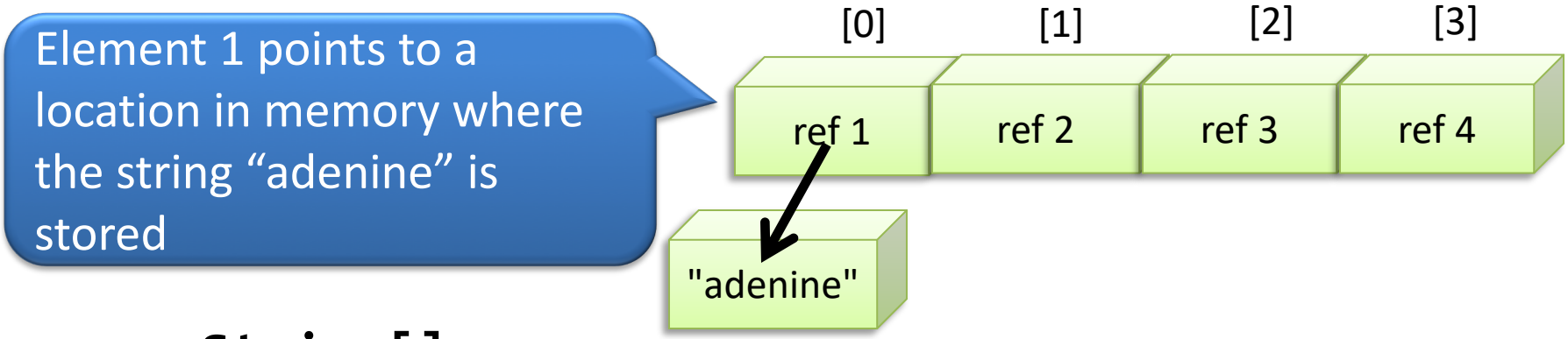

```
String[] nucs =
{"adenine", "cytosine", "guanine", "thymine"};
```

Primitive arrays

- An array of primitives will actually contain the primitive **values**.



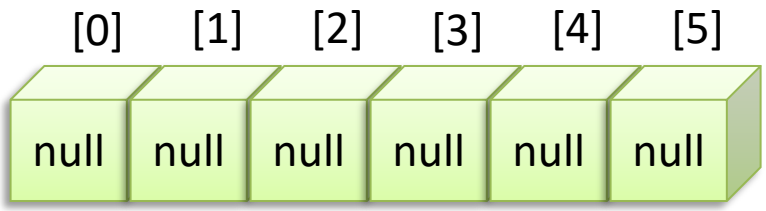
- An array of objects will hold the **references** to the objects, not the objects themselves




```
String[] nucs =
{"adenine", "cytosine", "guanine", "thymine"};
```

Object arrays

```
Cell[] cells = new Cell[6];
System.out.println("cell 2 has size: " +
cells[1].size);
```




```
$ java ArrayDemo
Exception in thread "main"
java.lang.NullPointerException
        at
ArrayDemo1.main(ArrayDemo1.java:32)
```

Object arrays are initialized with ***null*** values.

Accessing a null value causes a **NullPointerException**

Array access in real life

- Arrays most often accessed using a ***for*** loop:

```
Cell[] cells = new Cell[number];
for(int i = 0; i < number; i++) {
    cells[i] = new Cell();
}
```

Command-line arguments

- Of course, you have already seen an array passing by:

```
public static main(String[] args)
```

String[] args is the array of **command-line arguments**

Processing command-line arguments

```
public static void main(String[] args) {
    for(int i = 0; i < args.length; i++) {
        System.out.println("argument "
            + i + " is " + args[i]);
    }
}
```

```
$ java ArgsArrayDemo foo bar bla
argument 0 is foo
argument 1 is bar
argument 2 is bla
```

class Arrays

- If you want to quickly print an array use class Arrays:

```
import java.util.Arrays;
…
int[] numbers = {1, 2, 3, 4, 5};
Arrays.toString(numbers)
```

```
$ java ArrayTest
[1, 2, 3, 4, 5]
```

Multidimensional arrays & matrices

- Of course, you can make arrays of arrays of arrays (etc) as well, as long as they are of the same type!

```
int[][][] y = new int[2][3][3];
y[0] = new int[][]{{1},{1}};

int[][] z = new int[5][5];
y[1] = z;
y[1] = new int[][]{};
y[1] = new int[][]{{1,2,3}};
```

PART 2: Operators

Operator types

- Operators can be **modifying**

```
count++;
```

- Operators can be **comparing**

```
count < maximum
```

Operators and operands

- Operators can have one (unary), two (binary) or three (ternairy) **operands** (variables on which they operate)

```
count++;

count < maximum

(age > 65 ? old=true : old=false);
```

Operators and precedence

- Operators can be ordered by their **precedence**, the order in which they will be evaluated

```
int x = 41;
int count = 1;
int y = x + count++;
System.out.println("count="+count+"; y="+y);
```

```
$java OperatorDemo
count=2; y=42
```

operator precedence

operator	description	asso-ciation
[], ., ()	index brackets, method call, grouping parentheses	L → R
++, --	postincrement, postdecrement	R → L
++, --	preincrement, predecrement	R → L
new, (type)	object creation, type cast	R → L
*, /, %	multiplication, division, remainder (modulo)	L → R
+, -, +	addition, subtraction, String concatenation	L → R
<, <=, >, >=, instanceof	less than, less than or equal, greater than, greater than or equal, type comparison	L → R
==, !=	value & reference equality, inequality	L → R
&&	conditional AND	L → R
||	conditional OR	L → R
?:	conditional ternary operator	L → R
=, +=, -=, *=, /=, %=	assignment operators	R → L

operators and precedence

- When in doubt about precedence, always use parentheses ()
- They always have the highest precedence (and improve readability a lot)

```
x = y + z * 2 / Math.PI;

x = y + ((z * 2) / Math.PI);
```

special operators

- There are a few operators worth giving some attention. These are the modulo (%), ternairy (?:), and assignment operators (=, +=, -=, *=, /=, %=)
- The others are pretty much self-explanatory

Assignment operators

- When you want to change the value of a variable, you can use these shorthand assignment operators

`x = x + 2;` is the same as `x += 2;`

`x = x + y;` is the same as `x += y;`

`x = x * 3;` is the same as `x *= 3;`

Modulo operator

- The modulo operator returns the remainder of a division:

```
x = 10 % 3
```
→ x is 1

```
x = 8 % 5
```
→ x is 3

- The modulo operator is often used to deal with periodicity:

```
if(x % 2 == 0){
   /*deal with even numbers*/
} else {/* odd numbers*/ }
```

Ternary operator

- The ternary operator is actually a condensed if..else block but can give more comprehensive code:

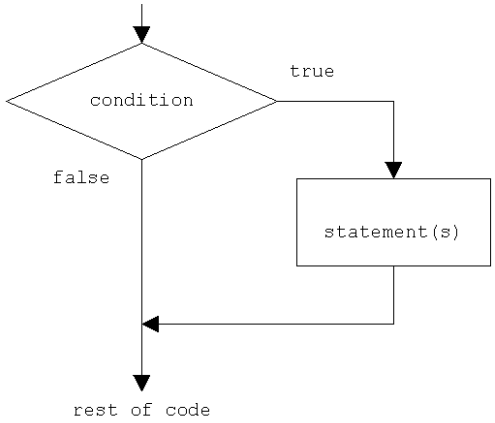
```
if (age < 18) {minor = true;}
else {minor = false;}
```

- can be replaced by:

```
minor = (age < 18 ? true : false);
```

PART 3: Control structures

if [else if] [else]



- Both "`else if`" and "`else`" are optional

```
if(condition){}
[else if(condition){}]*
[else{}]]?
```

condition is any Java expression evaluating to a boolean value
* means zero to many
? means zero or one

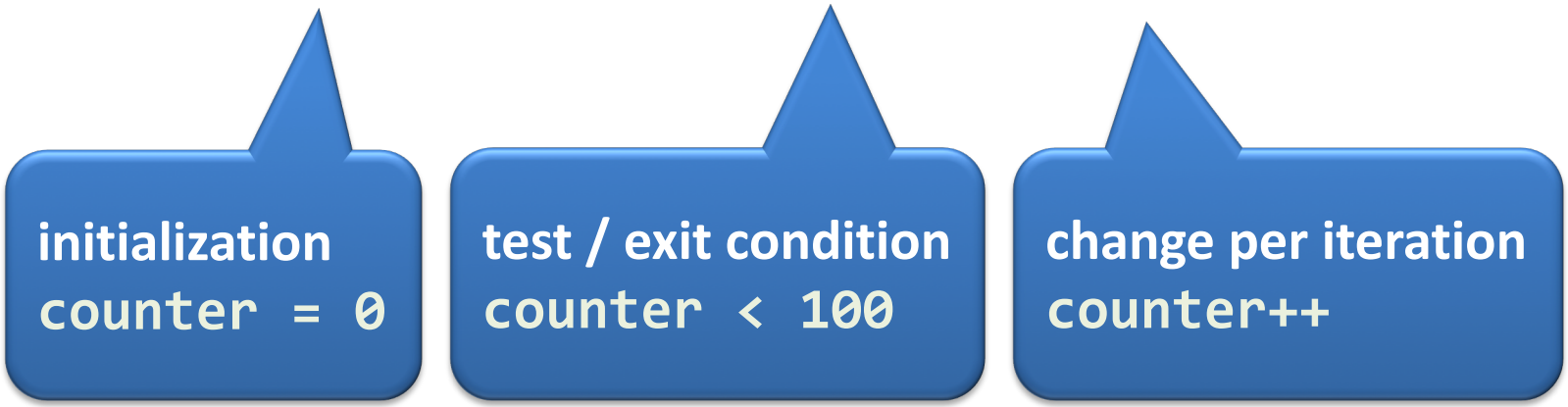
while

- While has two variants:
  - **`while (condition) {}`**
  - **`do {} while (condition)`**
- The difference is that do{}while() is guaranteed to be executed at least once

Control structures: *for*

- Iterates over a collection or a defined series of steps

```
for(init; condition; change){ }
```



Control structures: *for*

```
for( int i = 1; i <= 100; i++ ){
    String append = "ste";
    if(i>1 && i<20 && (i!=8) ) append = "de";
    System.out.println( "Ik heb een potje met vet, al op de tafel gezet, " +
            "ik heb een potje potje potje potje veeeeheeet al op de tafel gezet\n" +
            "Dit was het " + i + append + " couplet" );
}
```

The devil is in the details…

```
$ java ControlStructuresDemo2
Ik heb een potje met vet, al op de tafel gezet, ik heb een
potje potje potje potje veeeeheeet al op de tafel gezet
Dit was het 1ste couplet
Ik heb een potje met vet, al op de tafel gezet, ik heb een
potje potje potje potje veeeeheeet al op de tafel gezet
Dit was het 2de couplet
...
Ik heb een potje met vet, al op de tafel gezet, ik heb een
potje potje potje potje veeeeheeet al op de tafel gezet
Dit was het 99ste couplet
Ik heb een potje met vet, al op de tafel gezet, ik heb een
potje potje potje potje veeeeheeet al op de tafel gezet
Dit was het 100ste couplet
```

for(each)

- The for loop has a variation called the **foreach** loop
- Use it when you are not interested in the counter

```
for(variable : collection){}
```

Declare a variable to traverse a collection with

This is the collection

switch/case

- Especially useful for choosing between different ***discrete*** options

```
int type = 1;

switch( type ){
    case 1:
        processType1();
        break;
    case 2:
        processType2();
        break;
    default:
        processOtherTypes();

}
```

Choose an action based on the value of a variable. This can only be a primitive, an enum (seen later) or a String

Break out, or ALL cases after the matching case will run!

```
$ java ControlStructuresDemo4
Processing type one...
```

switch/case

- Switch/case is ***fall-through*** unless you break!

```
String s = "foo";
switch(s){
    case "foo": System.out.println("foo!");
    case "bar": System.out.println("bar!");
    case "baz": System.out.println("baz!");
    default: System.out.println("something else");
}
```

```
$ java ControlStructuresDemo4
foo!
bar!
baz!
something else
```

Breaking switch/case

```
String s = "foo";
switch(s){
    case "foo": System.out.println("foo!");
    case "bar": System.out.println("bar!"); break;
    case "baz": System.out.println("baz!"); break;
    default: System.out.println("something else");
}
```

```
$ java ControlStructuresDemo4
foo!
bar!
```

PART 4: Methods

- Organize code in chunks that serve a single well-defined goal (SRP!)
- Methods are pieces of code that often serve as "black box": you don't need to know how they work, as long as you know what goes in and what comes out

What are methods?

- Methods define the **behavior** of an object, modified by its state
- Methods can serve the inner workings of an object unseen, or be available to the outside world: this is something you decide upon (and a very important design aspect!)
- Some methods live outside object scope and are only available as class methods

Method signature

- Every method has a **signature**
- In Java, this is a

  binding contract
- The signature defines what can go in, what comes out, and who can use the method

Method anatomy

public means any code within the application can access (use) this method

int specifies the return type of this method

```
public int addInts(int x, int y) {
   int result = x + y;
   return result;
}
```

Between the parentheses you find the -typed- method parameters

Here, the contract is lived up to: an integer is returned to the caller

Parameters versus arguments

- A ***parameter*** is the variable which is part of the method's signature (method declaration)
- An ***argument*** is an expression used when calling the method

Method signature and body

In red is the method **signature**

```
public int addInts(int x, int y) {
   int result = x + y;
   return result;
}
```

In blue is the method **body**

Calling a method

```
public class MyMathTools{
  public int addInts(int x, int y){
    return (x + y);
  }
  /*test the math function*/
  public static void main(String[] args){
    MyMathTools mt = new MyMathTools();
    int result = mt.addInts(2, 10);
    mt.addInts(2, 10);
  }
}
```

You don't HAVE to catch what is returned, but then it is of course useless to call the method

How about cheating?

```
public class MyMathTools{
  public int addInts(int x, int y) {
    return (x + y);
  }
  /*test the math function*/
  public static void main(String[] args){
    MyMathTools mt = new MyMathTools();
    mt.addInts(3);
    mt.addInts(1, 2, 3);
    mt.addInts(2, "foo");
  }
}
```

These won't compile: you have to pass **exactly** two variables of type integer.

How about cheating?

```
public class MyMathTools{
   public int addInts(int x, int y){
      x + y;
      return;
   }
}
```

This **won't** compile: you have to return an int.

Contract flexibility

This **won't** compile:
LOSS OF PRECISION

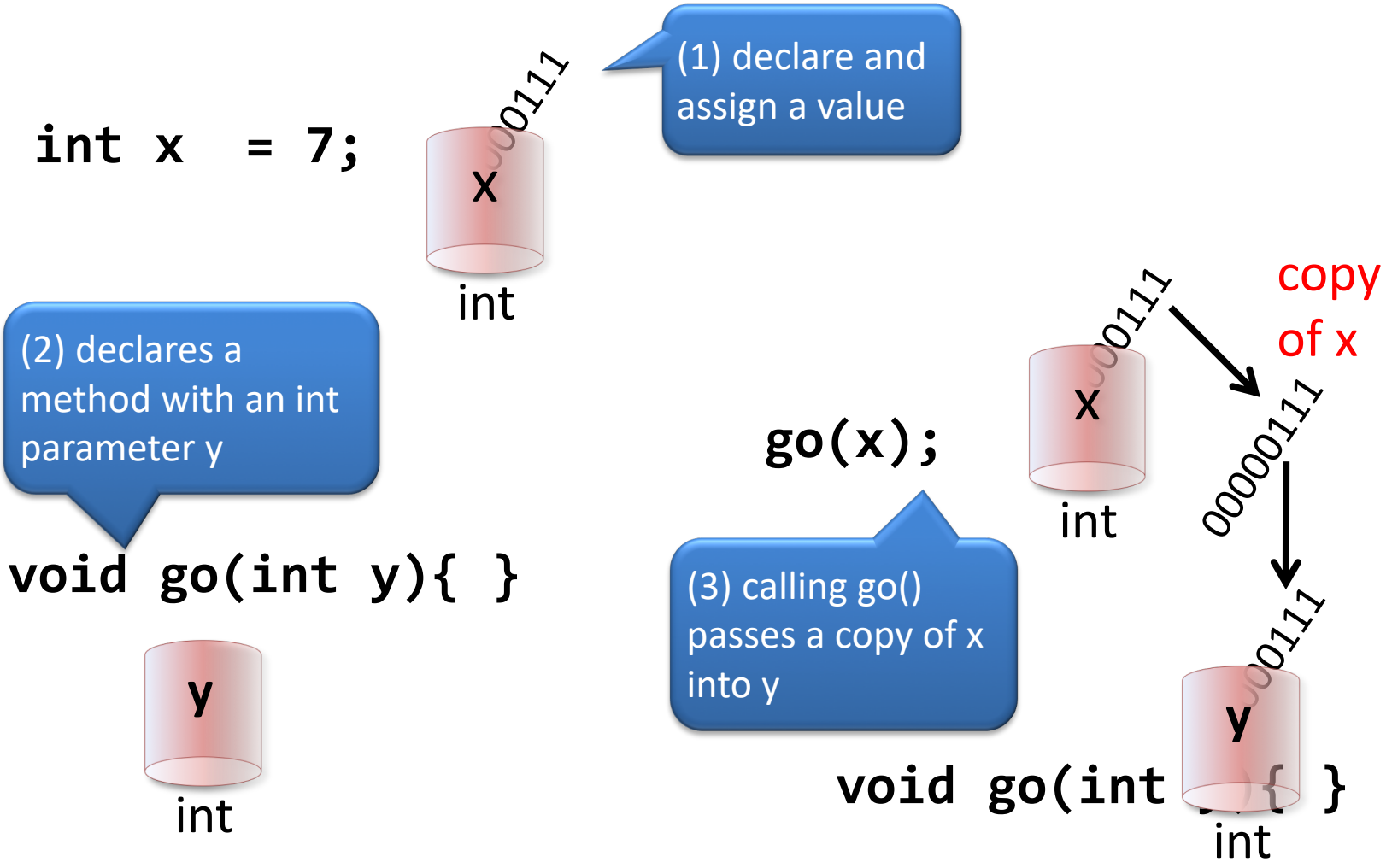
```
public void test() {
   System.out.println("adding ints: " +  addInts(42, 42));
   System.out.println("adding ints: " +  addInts(42.0, 42.0));
   System.out.println("adding doubles: "
                           +  addDoubles(42.0, 42.0));
   System.out.println("adding doubles: "
                           +  addDoubles(42, 42));
}
public int addInts(int x, int y){
   return x + y;
}
public double addDoubles(double x, double y){
   return x + y;
}
```

This **will** compile: NO
LOSS OF PRECISION

Passing arguments

- Java is pass-by-value
- This means: pass-by-copy
- Passing a reference means: pass a copy of the reference
- This reference copy will point to the same object as the original reference!

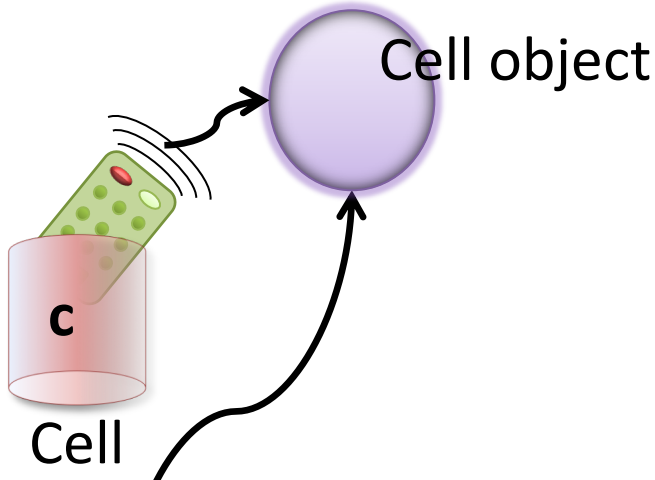
passing primitives: passing copies



passing object references

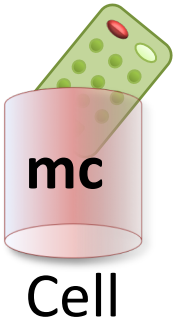
(1) An object is created and linked to reference variable c

```
Cell c = new Cell();
```

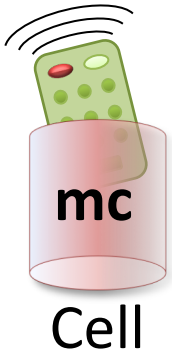



```
void go(Cell mc){ }
```

(2) Method go declares a para-meter of type Cell




```
go(c);
```



(3) go() is called with variable c. It (the reference!) is copied and local variable mc now also **points to the same Cell object**

What about multiple return values?

- Forget it! There is NO way you can return multiple values from a method.
- Unless…you put these values in a composite form

```
double[] doStatistics( int x, int y){
   int sum = x + y;
   double average = ((double)x + (double)y)/2;
   double[] result = {(double)sum, average};
   return result;
}
```

This is legal but not very nice: you coerce an int into a double where this is not appropriate

Returning multiple values

!!! THINK OBJECTS !!!

```
public class Statistics{
   int sum;
   double average;
}
```

Define a Statistics class that will hold your heterogenous data

```
Statistics doStatistics( int x, int y){
   Statistics stats = new Statistics();
   stats.sum = x + y;
   stats.average = ((double)x + (double)y)/2;
   return stats;
}
```

PS this is still not the nicest OO design but I hope it gets the point through

Can you specify default values?

- In Python and other languages, this is a much-used way to define methods:

```
>>> def divide(numerator, denominator = 2):
        return numerator/denominator

>>> divide(10, 5)
2
>>> divide(10)
5
```

This means: if no second argument is passed, use the default value of 2

Can you specify default values?

- In Java you have to use another technique: ***method overloading***:

```
double divide(double numerator, double denominator) {
    return numerator / denominator;
}
double divide(double numerator) {
    return divide(numerator, 2);
}
```

```
divide(10, 5);
divide(10);
```

Overloaded methods is the Java way of making different implementations of the same functionality

Method overloading

- Overloaded methods is the Java way of making different implementations of the same functionality
- It is a bit of a hassle, but more versatile at the same time, since this not only supports **default values** but also **extra pieces of algorithm code on top of default behavior**