

Parallel Programming

Today: A bit of JVM and Java recap connected to concurrency/parallelism

Why Java?

Widely used programming language in academia and industry

Lots of courses downstream use Java

Lots of resources available online and in books

Sophisticated support of concurrency in the language and in libraries

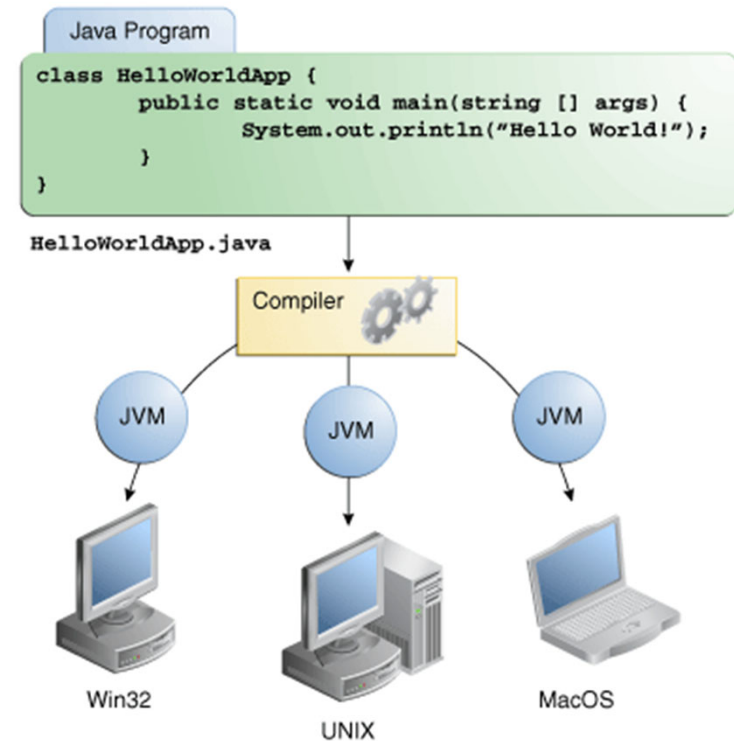
Java

Platform independence via bytecode interpretation

Java programs run (**in theory**) on any computing device (PC, mobile phones, Toaster, Windows, Linux, Android)

Java compiler translates source to byte code

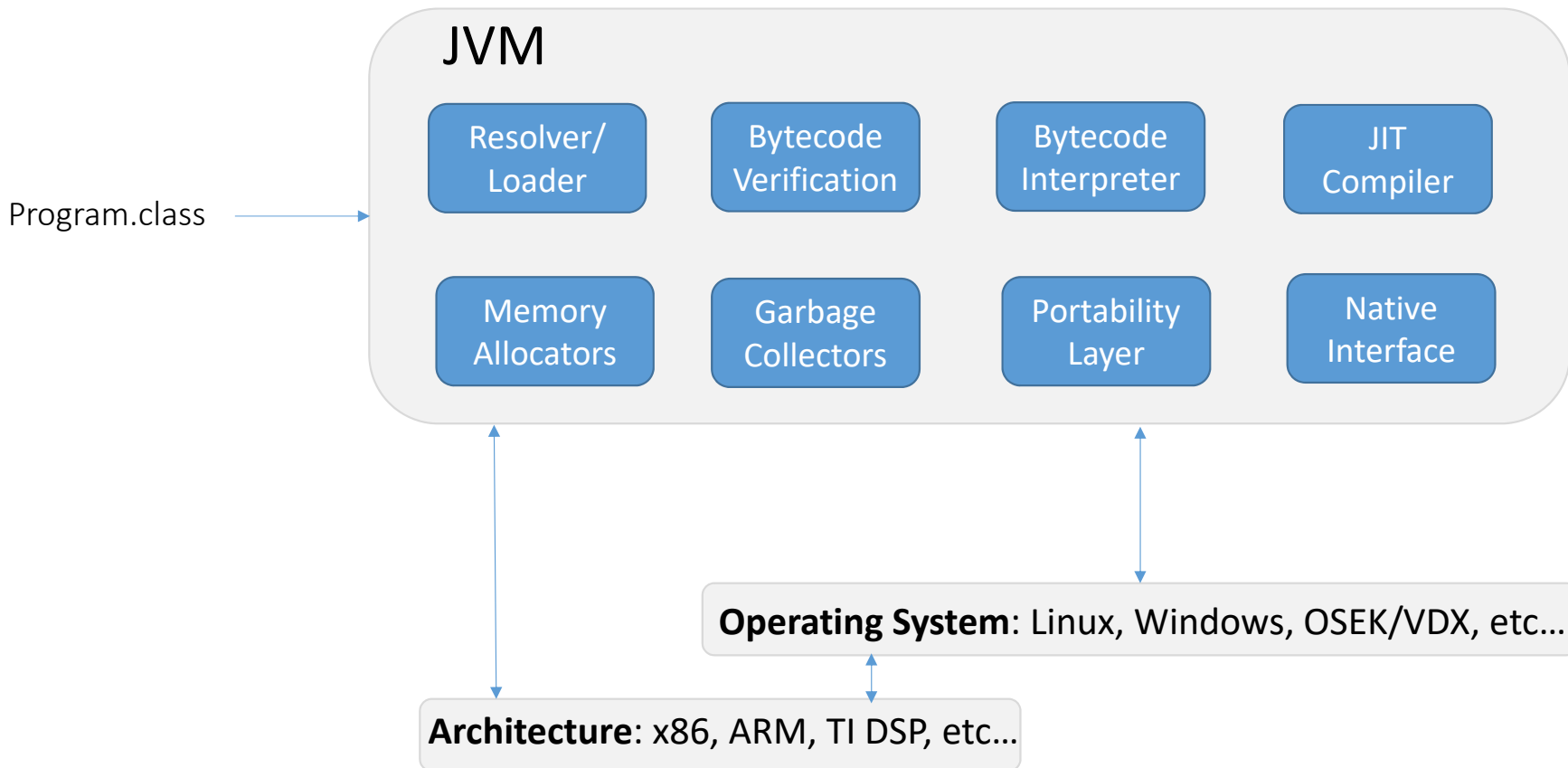
Java virtual machine (JVM) interprets the bytecode of the compiled program



Today

- Sneak peek into JVMs
 - Brief overview of JVM components ...
 - ... spending a bit more time on bytecode
 - This material is not examinable. Do not worry if you do not follow some of the concepts. This is only meant to give you a feel for the bigger picture (and can also be useful when debugging).
- Recap of Java
 - Certain constructs and patterns that are connected to concurrency
 - More recap slides in the slide deck, not covered in class

Key JVM Components



Resolver, Loader

Loads class files and setups their internal memory....but **when?**

```
class Test {  
  
    public static void main(String args[]) {  
        if (args[0].equals("nuf si HTE"))  
            LoadMe t = new LoadMe();  
    }  
}  
  
class LoadMe() {  
    static int x = 0;  
    static { System.out.println ("Got statically initialized"); }  
}
```

The JVM has a choice here:

Eager: the JVM resolves the reference to LoadMe when class Test is first loaded.

Lazy: the JVM resolves the reference to LoadMe when it is actually needed (here, when the LoadMe object is created). Most JVMs are lazy.

Static initialization of the class is quite **non-trivial**, can be done **concurrently by many Java threads** (we will see what a Thread is later). Typically done before the class is used.

Bytecode Verification

Automatically verifies bytecode provided to JVM satisfies certain security constraints. Usually done right after the class is located and loaded, but before static initialization

- bytecodes type check
- no illegal casts
- no conversion from integers to pointers
- no calling of directly private methods of another class
- no jumping into the middle of a method
-and others

Minor problem:

Automated verification is **undecidable**.

Practically, this means the verifier **may reject valid programs** that actually do satisfy the constraints. ☹️

The goal is to design a verifier that **accepts as many valid programs** as possible. 😊

Bytecode Interpreter

A program inside the JVM that interprets the bytecodes in the class files generated by the javac compiler using a **stack** and **local variable** storage.

- JVM is a **stack based abstract machine**: bytecodes pop and push values on the stack
- A set of registers, typically used for local variables and parameters: accessed by load and store instructions
- For each method, the number of stack slots and registers is specified in the class file
- Most JVM bytecodes **are typed**.

The bytecode interpreter is **typically slow** as its pushing and popping values from a stack...

One can speed-up the interpreter but in practice parts of code that are **frequently executed** simply get compiled by the Just in Time (JIT) compiler to native code (e.g., Intel x86, etc)...next.

Just-In-Time Compiler (JIT)

Compiles the bytecode to **machine code** (e.g., ARM, DSP, x86, etc) **on-demand**, especially when a method is frequently executed (hot method). JIT makes bytecodes fast 😊

```
class Test {  
  
    public static int inc(int j) {return j + 1; }  
  
    public static void main(String args[]) {  
        int j = 0;  
        for (int i = 0; i < 100000; i++)  
            j = inc(j);  
    }  
}
```

Compilation of bytecode to machine code happens **during program execution**. Typically needs **profiling data** to know which method is hot or cold. **Can be expensive to gather** during execution.

In this example, method `inc(int)` is a hot method so may be **inlined** inside the main to avoid the overheads of function calling.

A modern JIT compiler has 100's of optimizations...

Memory Allocators

Consists of, often **concurrent algorithms**, which are invoked when your Java program allocates memory.

```
class Test {  
  
    public static void main(String args[]) {  
  
        A a = new A();  
  
        int t[][] = new int[2][5];  
    }  
}
```

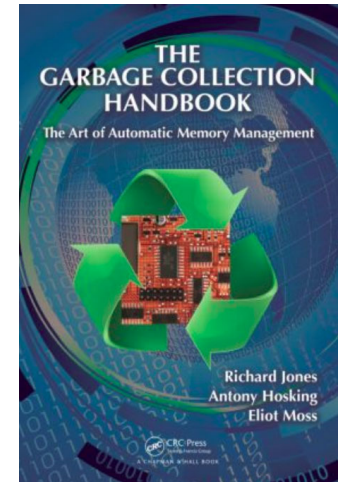
Object allocation in Java invokes the JVM memory allocator. The JVM memory allocator often has to ask the underlying OS for memory which it then **manages internally**.

The allocator algorithms typically **have to be concurrent** because multiple Java threads (we will learn about threads later) can allocate memory. Otherwise, if **sequential**, one may see major pause times in their application.

Garbage Collectors (GC)

JVM uses many different GC algorithms, often **concurrent and parallel**, invoked periodically to collect memory unreachable by your program.

```
class Test {  
  
    public static void main(String args[]) {  
        A a = new A();  
        a = null; // memory of A() now unreachable.  
        System.gc(); // force GC  
        while(true); // wait till it triggers ☺  
    }  
  
    class A() {  
        public void finalize() {  
            System.out.println("I got freed by the GC");  
        }  
    }  
}
```



Frees the programmer from having to free memory manually...which is good as it avoids tricky bugs.

Many different GC algorithms: generational, concurrent, parallel, mark and sweep, etc. Trade-off different performance goals. Concurrent GC algorithms are **very difficult to get correct**.

finalize() method called when GC collects the object.

Native Interface

When your Java program calls a **native** method, one has to convert the JVM parameters (e.g., what is on the stack) into machine registers (e.g, x86) following the calling convention.

```
class Test {  
  
    public static native int print(double d);  
  
    public static void main(String args[]) {  
        print(5.2);  
    }  
}
```

As the JVM interpreter is executing the Java program, at some point it may have to call a native method: some code written in C or C++ say.

To do so, the JVM has to pass the parameters to the native method in a particular way so to interact with binaries on the particular platform.

This is not a large module but can be tricky to get the types correct.

java.lang.Object contains many native methods (e.g., starting a thread) for which the JVM provides an **internal** implementation.

Portability Layer

When the JVM is to run on top of Windows vs. Linux or x86 vs. ARM, the JVM designer must implement a small number of JVM constructs (e.g., **synchronization, threading**) using the primitives of the underlying operating system and architecture.

e.g.: Java notion of a Thread

must be mapped to

Portability
Layers

e.g.: OS notion of a thread

Operating System: Linux, Windows, OSEK/VDX, etc...

Example: Java provides **its own notion** of a thread (as **we will see later**). However, the operating system has a different notion of what a thread is. So this layer somehow needs to come up with a way to use the OS notion of a thread to provide the Java notion of a thread that the Java programmer expects.

If you want your JVM to run on a different OS, you need to write the portability layer.

Lets look a bit at the bytecodes...

```
class Test {
```

```
    static int x = 2018;  
    double d;
```

```
    public static native int print(double d);  
    public double pp(int a) { return a; }
```

```
    public static void main(String args[]) {  
        Test t = new Test();  
        t.d = t.pp(1) + x;  
        Test.print(t.d);  
    }  
}
```

javac Test.java



Test.class

What is inside
here?

Look inside via: `javap -c Test`

This kind of usage of 'javap' is **not examinable** but it may help you to get deeper understanding of how the Java language actually gets executed.

We will see how this is **helpful later** with constructs such as synchronized and volatile and why this is instructive.

You can find the meaning of all JVM instructions here:

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5>

```
class Test {  
    static int x;  
    double d;
```

Constructor for class Test

```
Test(),
```

```
Code:
```

```
0: aload_0
```

```
1: invokespecial #1 // Method java/lang/Object."<init>": ()V
```

```
4: return
```

```
public static native int print(double);
```

```
public double pp(int);
```

```
Code:
```

```
0: iload_1
```

```
1: i2d
```

```
2: dreturn
```

```
static {}; JVM invokes this code before main()
```

```
Code:
```

```
0: sipush    2018
```

```
3: putstatic #5 // Field x:I
```

```
6: return
```

```
}
```

Pushes content of local variable 0 (note: the variable is of a reference type) to the stack.

Invoke constructor for the superclass of Test, that is, java.lang.Object...and clear the stack.

Native method. Its implementation could be provided for example in a C/C++ library.

Pushes content of local variable 1 (type integer) to stack.

convert the integer on the stack to a double.

Pop value from stack and return it.

push constant 2018 of type short (hence: si) to stack

pop 2018 from stack and write it to static field x.


```
public static void main(java.lang.String[]);
```

Code:

```
0: new      #2      // class Test
3: dup
4: invokespecial #3    // Method "<init>":()V
7: astore_1
8: aload_1
9: aload_1
10: iconst_1
11: invokevirtual #4    // Method pp:(I)D
14: getstatic  #5      // Field x:I
17: i2d
18: dadd
19: putfield  #6      // Field d:D
22: aload_1
23: getfield  #6      // Field d:D
26: invokestatic #7    // Method print:(D)I
29: pop
30: return
```

Create the object of class Test and push it on the stack.
Object **not yet initialized**! Triggers the JVM's memory allocator.

Duplicate object reference on the stack.

Invoke constructor for top-of-stack object (pops it). **This initializes** it as we saw before. After, 1 reference to object remains

Store top-of-stack reference in local variable and pops it. Stack now empty.

Load reference from local variable 1 onto the stack.

Load reference from local variable 1 onto the stack again.

push the constant 1 onto the stack

Invoke method pp(). Pops the constant 1 and the reference. Method returns a value stored on top of the stack.

Read the value of the static field 'x' and push it onto the stack.

Convert to a double.

Add two values on top of stack (pop them) and produce 1 value.

.....// we skip 19, 22 and 23 here: can do it yourself.

We are invoking a **native method**! The native interface component of the JVM will make sure local vars/stack information is converted to the particular binary interface...

Today

- Sneak peek into JVMs
 - Brief overview of JVM components ...
 - ... spending a bit more time on bytecode
 - This material is not examinable. Do not worry if you do not follow some of the concepts. This is only meant to give you a feel for the bigger picture (and can also be useful when debugging).
- Recap of Java
 - Certain constructs and patterns that are connected to concurrency
 - More recap slides in the slide deck, not covered in class

Structure of a Java program

```
public class name {  
    public static void main(String[] args) {  
        statement;  
        statement;  
        ...  
        statement;  
    }  
}
```

The diagram illustrates the structure of a Java program with three callouts:

- class:** a program (points to the `name` in `public class name`)
- method:** a named group of statements (points to the `main` method signature)
- statement:** a command to be executed (points to one of the statements inside the `main` method)

Every executable Java program consists of a **class** that contains a **method** named `main`, that contains the **statements** (commands) to be executed.

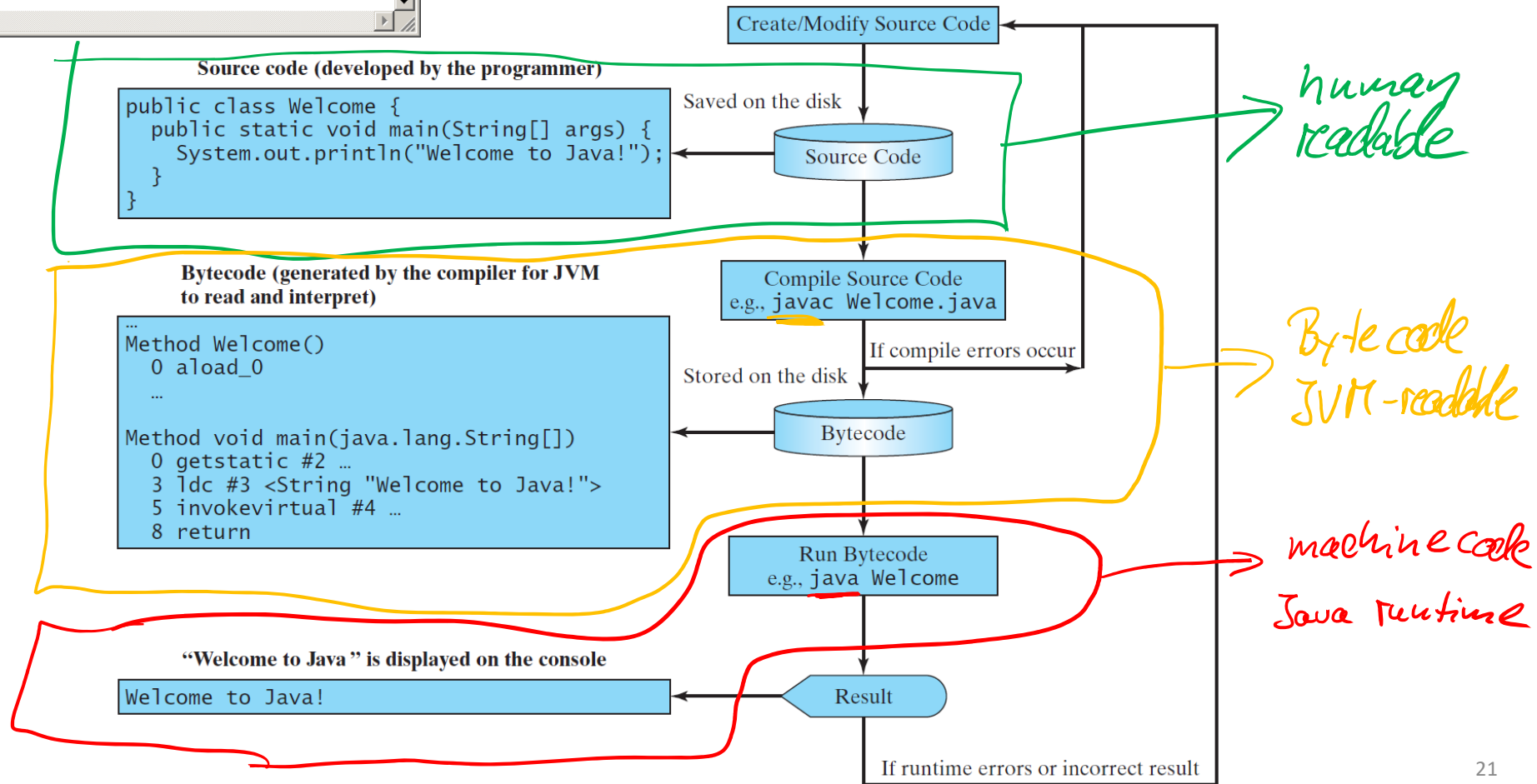
Keywords

Keyword: An identifier that you cannot use (to name methods, classes, etc) because it already has a reserved meaning in Java.

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	...
continue	goto	package	synchronized	

```
Welcome - Notepad
File Edit Format View Help
// This application program prints Welcome to Java!
public class Welcome {
    public static void main(String[] args) {
        System.out.println("Welcome to Java!");
    }
}
```

Creating, Compiling, and Running Programs



Different kinds of errors

1. Compiler errors
2. Runtime errors
3. Logic errors

Compiler error example

```
1 public class Hello{
2     public static void main(String[] args) {
3         System.out.println("Hello, world!");
4     }
5 }
```

Compiler output:

Hello.java:2: error: invalid method declaration; return type required
public static [^]main(String[] args) {

Hello.java:3: ';' expected
}

2 errors

- The compiler shows the line number where it found the error.
- The error messages can be tough to understand!

Runtime error example

```
public class ShowRuntimeErrors {  
    public static void main(String[] args) {  
        System.out.println(1 / 0);  
    }  
}
```

Exception in thread "main"

java.lang.ArithmeticException: / by zero

at

ShowRuntimeErrors.main>ShowRuntimeErrors.java:4)

Logic error example

Celsius to Fahrenheit:
 $F = 1.8 * C + 32$

```
public class ShowLogicErrors {  
    public static void main(String[] args) {  
        System.out.println("35 Celsius is Fahrenheit");  
        System.out.println((9 / 5) * 35 + 32);  
    }  
}
```

```
c:\book>java ShowLogicErrors  
35 Celsius is Fahrenheit  
67
```



What is going on here?

```
javap -c showlogicalerrors
```

```
....  
8:  getstatic   #2          // Field java/lang/System.out:Ljava/io/PrintStream;  
11:  bipush     67  
13:  invokevirtual #5        // Method java/io/PrintStream.println:(I)V  
....
```

Logic Error Example

Celsius to Fahrenheit:
 $F = 1.8 * C + 32$

```
public class ShowLogicErrors {  
    public static void main(String[] args) {  
        System.out.println("35 Celsius is Fahrenheit");  
        System.out.println((9.0 / 5.0) * 35 + 32);  
    }  
}
```

```
c:\book>java ShowLogicErrors  
35 Celsius is Fahrenheit  
95
```

How about now?

javap -c showlogicalerrors

```
....  
8: getstatic    #2          // Field java/lang/System.out:Ljava/io/PrintStream;  
11: ldc2_w      #5          // double 95.0d  
14: invokevirtual #7          // Method java/io/PrintStream.println:(D)V  
....
```

What in the world is ldc2_w!? Where is the number 95.0 which is to be printed?!

Lets look at the constant pool of the file...

```
javap -verbose showlogicalerrors
```

```
...  
Constant pool:  
  #1 = Methodref      #9.#18      // java/lang/Object."<init>":()V  
  #2 = Fieldref       #19.#20     // java/lang/System.out:Ljava/io/PrintStream;  
  #3 = String         #21         // 35 Celsius is Fahrenheit  
  #4 = Methodref      #22.#23     // java/io/PrintStream.println:(Ljava/lang/String;)V  
  #5 = Double         95.0d  
....
```

The constant **95** was just recorded in the file in a place called the **constant pool** at location 5 and the **ldc2_w** instruction just loads it from location 5 onto the stack, at runtime.

Data types

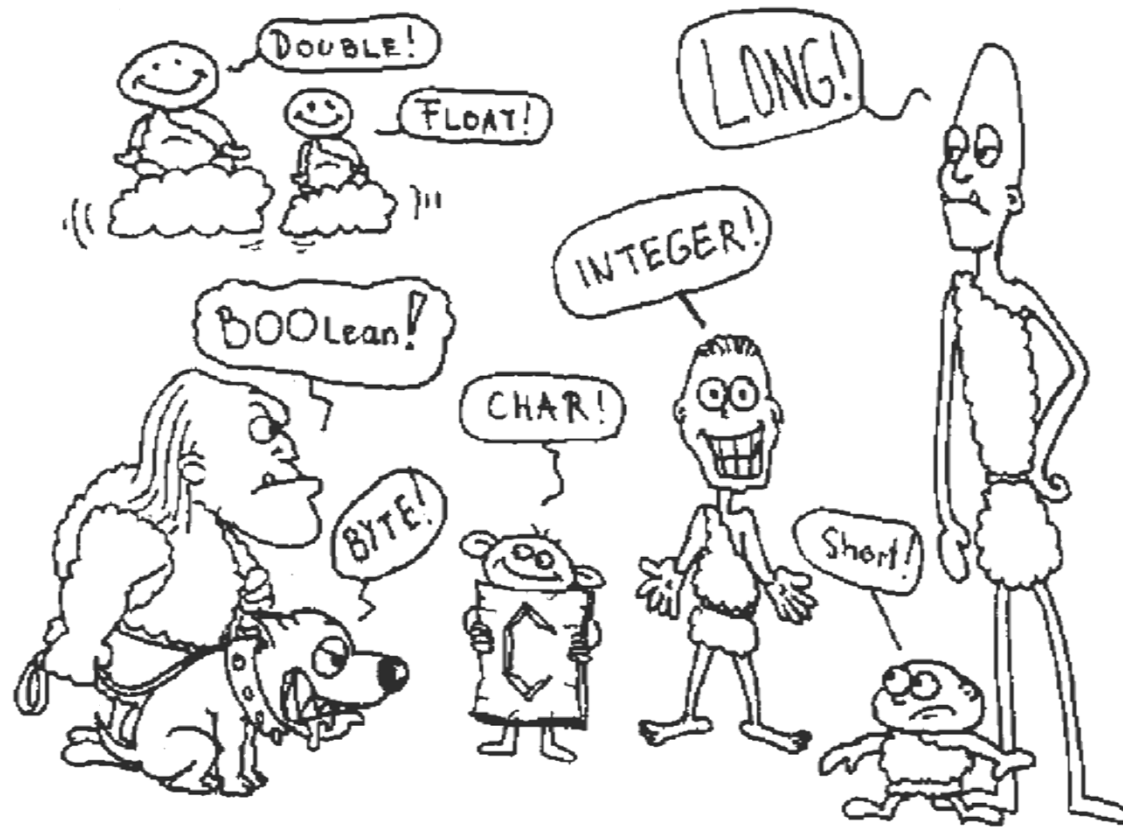
Type: A category or set of data values.

- Constrains the operations that can be performed on data
- Many languages ask the programmer to specify types
- Examples: integer, real number, string

Internally, computers store everything as 1s and 0s

104 → 01101000

Primitive Types in Java



Source: Christian Ullenboom: *Java ist auch eine Insel*,
Galileo Computing, 8. Auflage, 2009, ISBN 3-8362-1371-4

Java's primitive types

Primitive types: 8 simple types for numbers, text, etc.

Name	Description	Examples
<code>int</code>	integers (up to $2^{31} - 1$)	<code>42, -3, 0, 926394</code>
<code>double</code>	real numbers (up to 10^{308})	<code>3.1, -0.25, 9.4e3</code>
<code>char</code>	single text characters	<code>'a', 'X', '?', '\n'</code>
<code>boolean</code>	logical values	<code>true, false</code>

Why does Java distinguish integers vs. real numbers?

Primitive Types – Complete List

Integer:

- **byte** (8 Bits)
- **short** (16 Bits)
- **int** (32 Bits)
- **long** (64 Bits)

Range: -2147483648 ... 2147483647
or -2^{31} to $2^{31}-1$

Real numbers:

- **float** (32 Bits)
- **double** (64 Bits)

Examples: 18.0 , -0.18e2 , .341E-2

Characters (Unicode):

- **char** (16 Bits)

65536 different values, allows for
non-English characters

Booleans:

- **boolean**

Values: **true** , **false**
Operators: &&, |, !

double vs. float

The double type values are more accurate than the float type values. For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays 1.0 / 3.0 is 0.3333333333333333



16 digits

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays 1.0F / 3.0F is 0.33333334



7 digits

Type conversion (Casting)

Java is a **strongly typed** language, so the compiler can detect type errors at compile time

```
int myInt;  
float myFloat = -3.14159f;  
myInt = myFloat;
```

Compile time error

```
int myInt;  
float myFloat = -3.14159f;  
myInt = (int)myFloat;
```

Explicit type cast (truncates value)

Bytecode:

```
0: ldc      #2          // float -3.14159f  
2: fstore_2  
3: fload_2  
4: f2i  
5: istore_1  
6: return
```

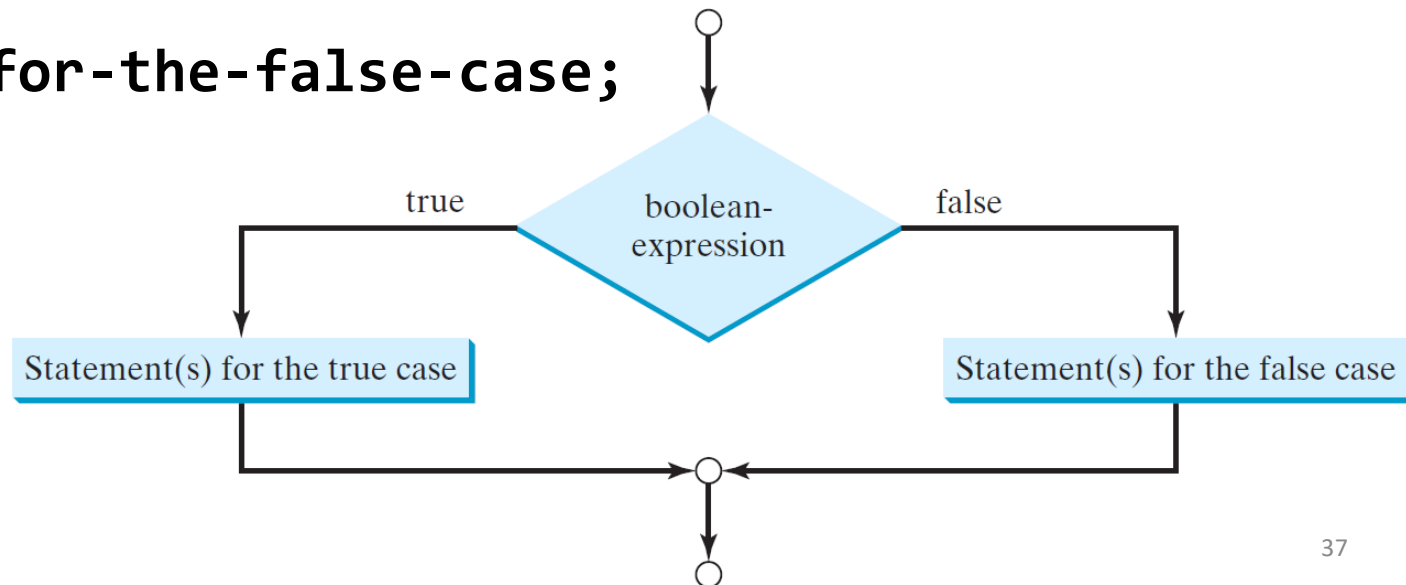
Type Casting - continued

Type casting can be useful/meaningful when dealing with references

```
Dog h; Animal rocky;  
  
if(rocky instanceof Dog)  
    h = (Dog)rocky;
```

The Two-way `if` Statement

```
if (boolean-expression) {  
    statement(s) - for-the-true-case;  
}  
else {  
    statement(s) - for-the-false-case;  
}
```



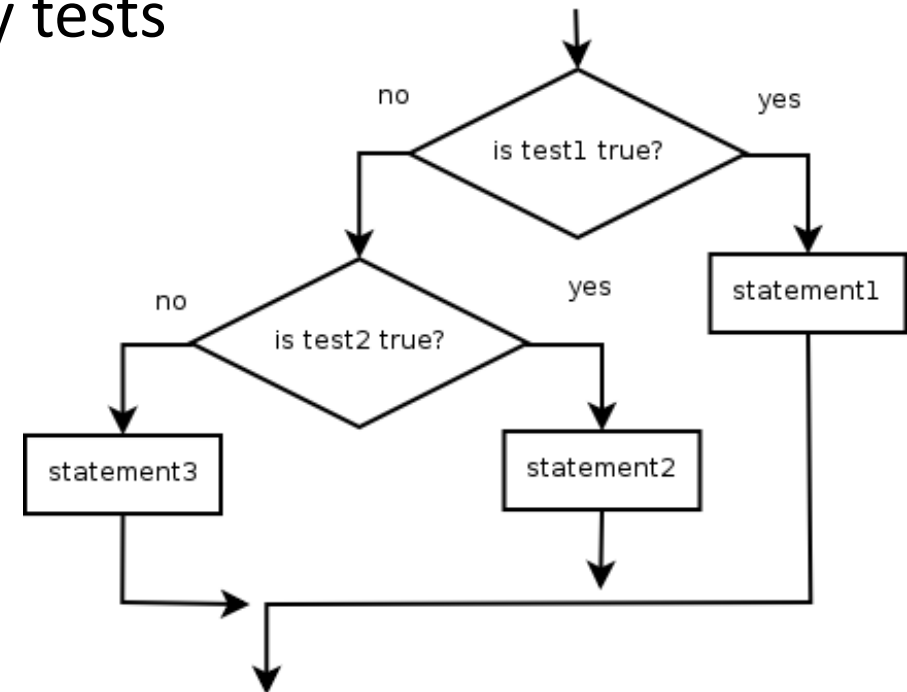
Nested if/else

Chooses between outcomes using many tests

```
if (test) {  
    statement(s);  
} else if (test) {  
    statement(s);  
} else {  
    statement(s);  
}
```

Example:

```
if (x > 0) {  
    System.out.println("Positive");  
} else if (x < 0) {  
    System.out.println("Negative");  
} else {  
    System.out.println("Zero");  
}
```



Tip: in parallelism/concurrency...try to have the if /else's read from a local variable.

switch Statements

```
switch (status) {  
    case 0: compute taxes for single filers;  
        break;  
    case 1: compute taxes for married file jointly;  
        break;  
    case 2: compute taxes for married file separately;  
        break;  
    case 3: compute taxes for head of household;  
        break;  
    default: System.out.println("Errors: invalid status");  
        System.exit(1);  
}
```

switch Statement Rules

The switch-expression must yield a value of char, byte, short, or int type and must always be enclosed in parentheses.

The value1, ..., and valueN must have the same data type as the value of the switch-expression. The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression.

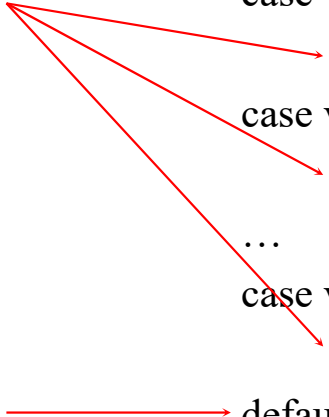
```
switch (switch-expression) {  
    case value1: statement(s)1;  
        break;  
    case value2: statement(s)2;  
        break;  
    ...  
    case valueN: statement(s)N;  
        break;  
    default: statement(s)-for-default;  
}
```


switch Statement Rules

The keyword break is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement. If the break statement is not present, the next case statement will be executed.

The default case, which is optional, can be used to perform actions when none of the specified cases matches the switch-expression.

```
switch (switch-expression) {  
    case value1: statement(s)1;  
        break;  
    case value2: statement(s)2;  
        break;  
    ...  
    case valueN: statement(s)N;  
        break;  
    default: statement(s)-for-default;  
}
```

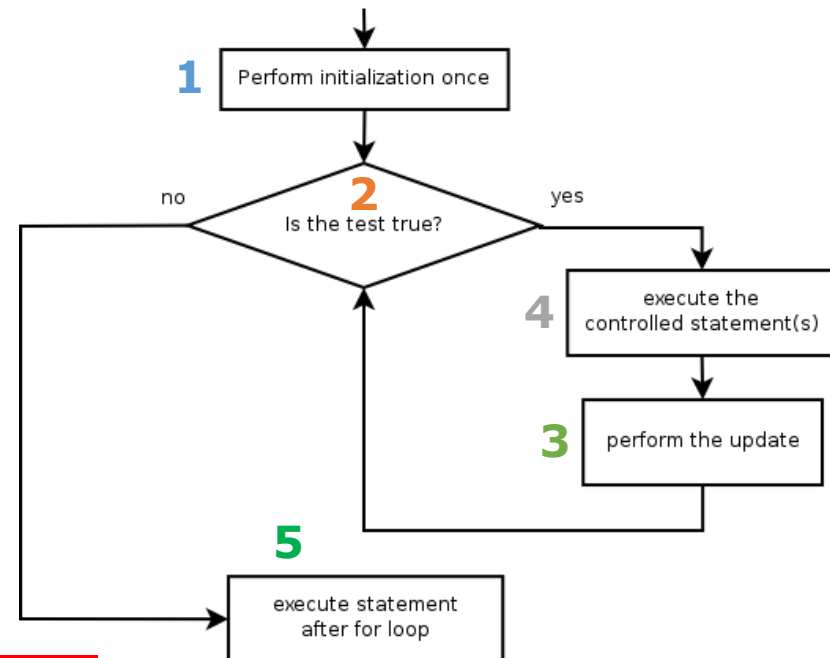


Loops: walkthrough

```
for (int i = 1; i <= 4; i++) {  
    System.out.println(i + " squared = " + (i * i));  
}  
System.out.println("Whoo!");
```

Output:

```
1 squared = 1  
2 squared = 4  
3 squared = 9  
4 squared = 16  
Whoo!
```



What can happen if 'i' is not a local variable and we have many processes? ☺

Categories of loops

Bounded loop: Executes a known number of times.

- The `for` loops we have seen are bounded loops.
 - print "hello" 10 times.
 - print each odd number between 5 and 127.

Unbounded loop: where number of times its body repeats is unknown in advance.

- e.g. repeat until the user types "q" to quit.

How would you write the mutual exclusion algorithm for a single participant from last lecture?
What kind of loop would we use?

The while loop

while loop: repeatedly executes its body as long as a logical test is true.

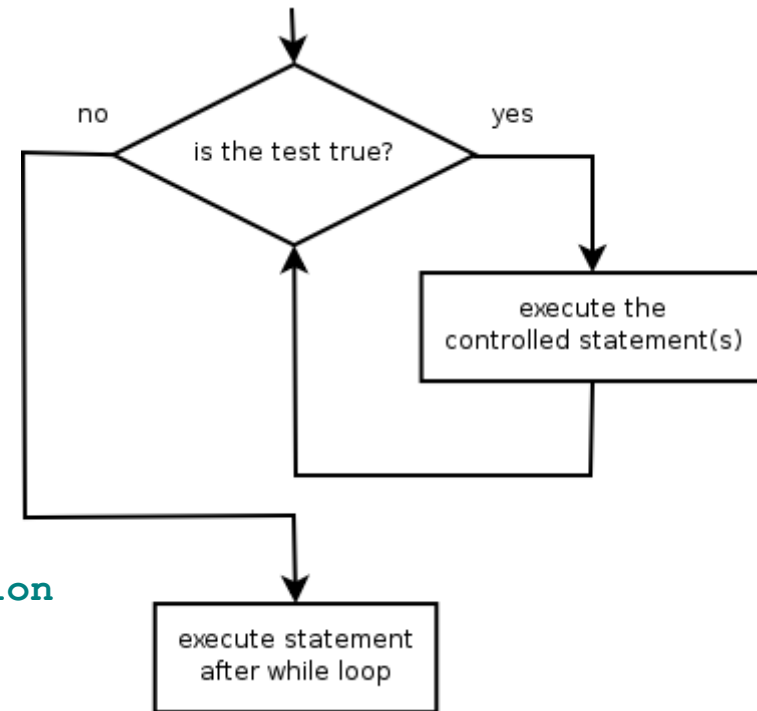
```
while (test) {  
    statement(s);  
}
```

Example:

```
int num = 1;  
while (num <= 200) {  
    System.out.print(num + " ");  
    num = num * 2;  
}
```

// output: 1 2 4 8 16 32 64 128

```
// initialization  
// test  
  
// update
```



Example while loop

```
// finds the first factor of 91, other than 1
int n = 91;
int factor = 2;
while (n % factor != 0) {
    factor++;
}
System.out.println("First factor is " + factor);
// output: First factor is 7
```

The do/while loop

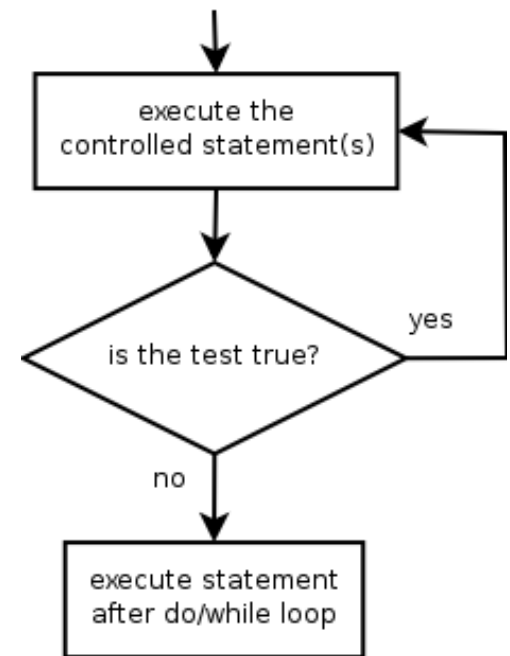
do/while loop: Performs its test at the *end* of each repetition.

- Guarantees that the loop's { } body will run at least once.

```
do {  
    statement(s);  
} while (test);
```

// Example: prompt until correct password is typed

```
String phrase;  
do {  
    System.out.print("Type your password: ");  
    phrase = console.next();  
} while (!phrase.equals("abracadabra"));
```



Arrays

Array: object that stores many values of the same type.

- **element**: One value in an array.
- **index**: A 0-based integer to access an element from an array.

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	12	49	-2	26	5	17	-6	84	72	3

element 0				element 4						element 9
-----------	--	--	--	-----------	--	--	--	--	--	-----------

Array declaration

type[] **name** = new **type**[**length**];

- Example:

```
int[] numbers = new int[10];
```

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>
<i>value</i>	0	0	0	0	0	0	0	0	0	0

Accessing elements

name [**index**] *// access*

name [**index**] = **value**; *// modify*

Example:

```
numbers[0] = 27;
```

```
numbers[3] = -6;
```

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>
<i>value</i>	27	0	0	-6	0	0	0	0	0	0

```
System.out.println(numbers[0]);
```

```
if (numbers[3] < 0) {
```

```
    System.out.println("Element 3 is negative.");
```

```
}
```

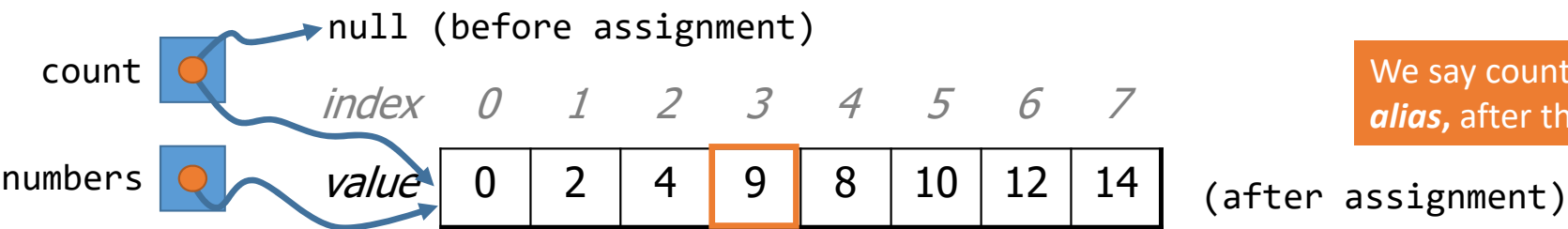
Arrays are reference types

```
int [] numbers = new int[8]; // new array of type integer and length 8

for (int i = 0; i < 8; i++) {
    numbers[i] = 2 * i;
}

for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
// output: 0 2 4 6 8 10 12 14

int [] count;
count = numbers;
count[3] = 9;
```



We say count and numbers *alias*, after the assignment

How are multi dimensional arrays represented?

Example: `int [] [] PP = new int[10][20]`

Strings

String: An object storing a sequence of text characters.

- Unlike most other objects, a `String` can be created without `new`.

```
String name = "text";  
String name = expression;
```

- Examples:

```
String name = "ETH 2019 Parallel Programming";  
int x = 3;  
int y = 5;  
String point = "(" + x + ", " + y + ")";
```

Indexes

Characters of a string are numbered with 0-based *indexes*:

```
String name = "R. Kelly";
```

index	0	1	2	3	4	5	6	7
character	R	.		K	e	l	l	y

- First character's index : 0
- Last character's index : 1 less than the string's length
- The individual characters are values of type `char`

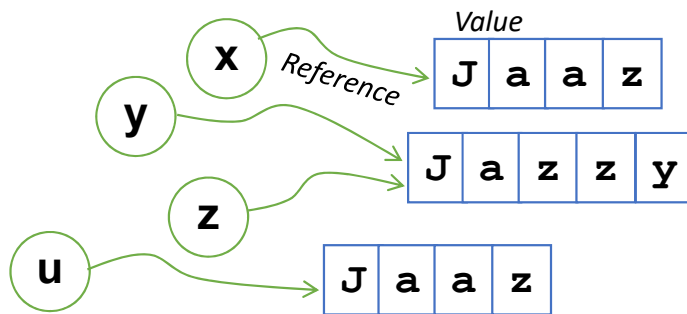
String methods

Method name	Description
<code>indexOf(str)</code>	index where the start of the given string appears in this string (-1 if not found)
<code>length()</code>	number of characters in this string
<code>substring(index1, index2)</code> or <code>substring(index1)</code>	the characters in this string from <i>index1</i> (inclusive) to <i>index2</i> (<u>exclusive</u>); if <i>index2</i> is omitted, grabs till end of string
<code>toLowerCase()</code>	a new string with all lowercase letters
<code>toUpperCase()</code>	a new string with all uppercase letters

These methods are called using the dot notation:

```
String gangsta = "Dr. Dre";  
System.out.println(gangsta.length());    // 7
```

Comparing strings: Examples



- `x == u` → „false“
- `x.equals(u)` → „true“
- `u.equals(x)` → „true“
- `y == z` → „true“
- `y.equals(z)` → „true“

Here, y and z *alias*,
while x and u don't

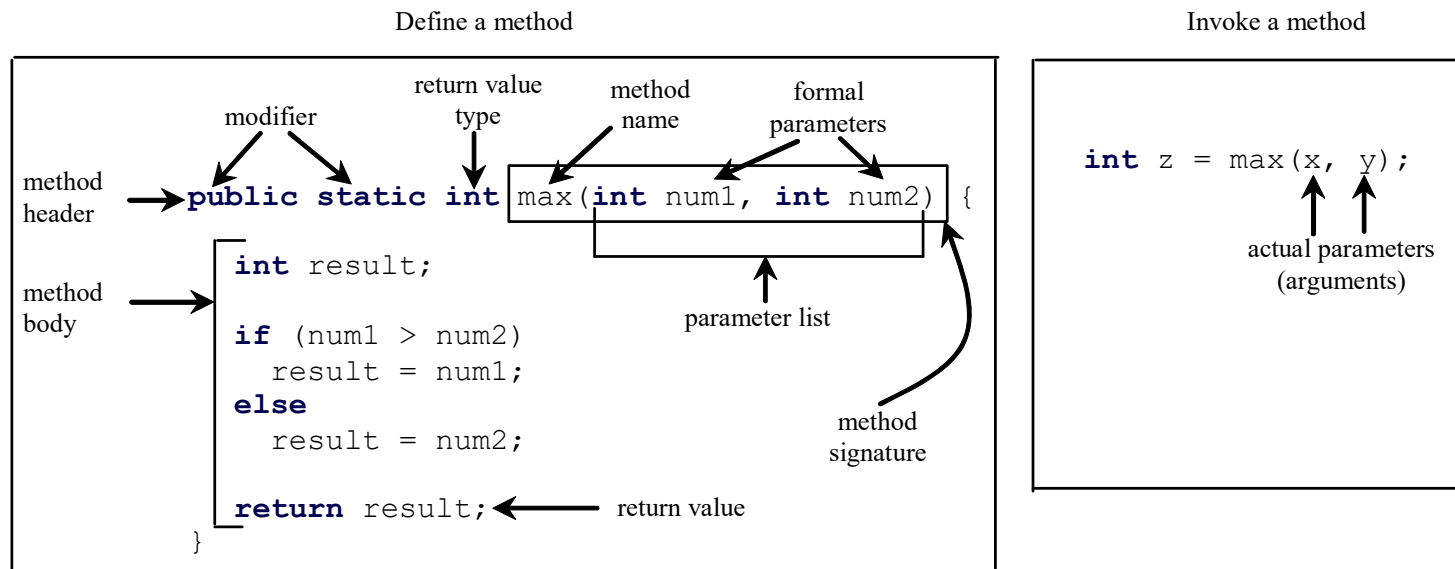
String test methods

Method	Description
<code>equals(str)</code>	whether two strings contain the same characters
<code>equalsIgnoreCase(str)</code>	whether two strings contain the same characters, ignoring upper vs. lower case
<code>startsWith(str)</code>	whether one contains other's characters at start
<code>endsWith(str)</code>	whether one contains other's characters at end
<code>contains(str)</code>	whether the given string is found within this one

```
String name = console.next();  
if (name.startsWith("Prof")) {  
    System.out.println("When are your office hours?");  
} else if (name.equalsIgnoreCase("STUART")) {  
    System.out.println("Let's talk about parallelism!");  
}
```

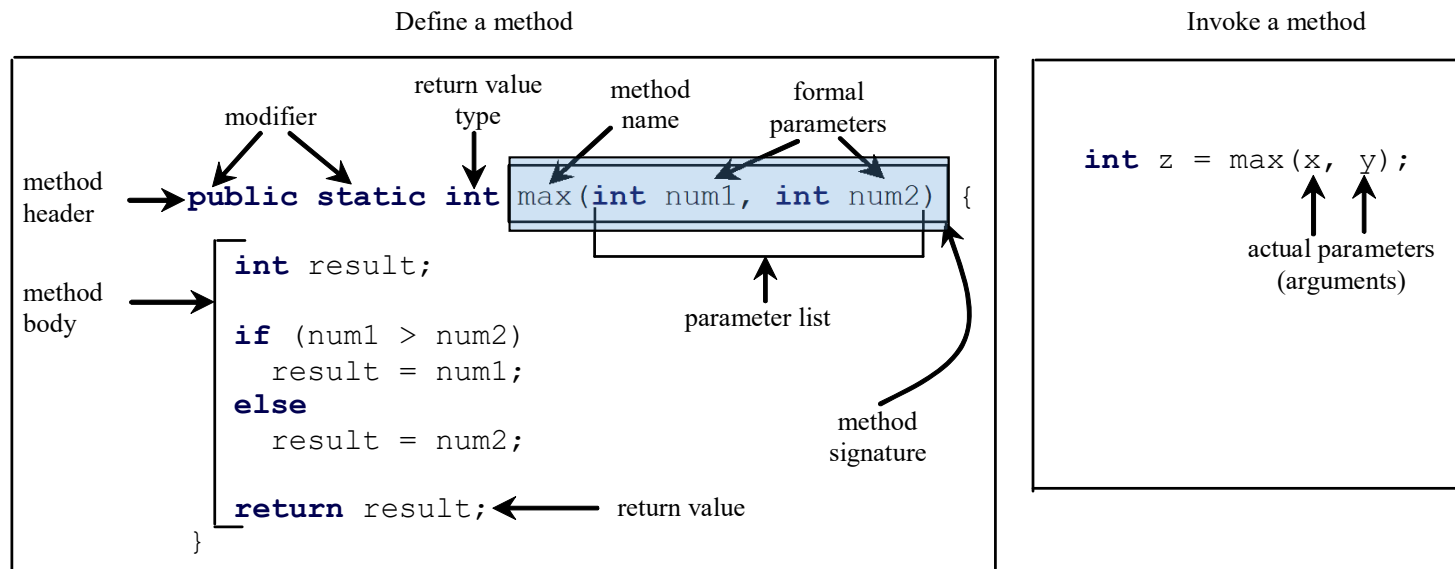

Defining Methods

A method is a collection of statements that are grouped together to perform an operation.



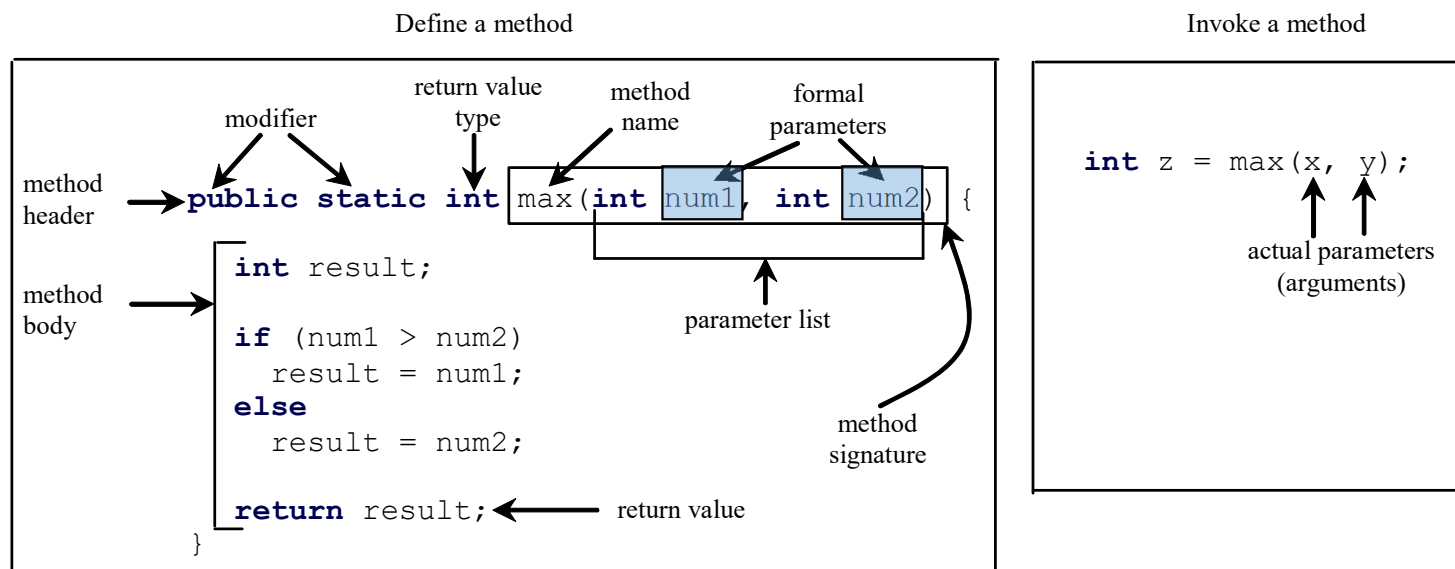
Method Signature

Method signature is the combination of the method name and the parameter list.



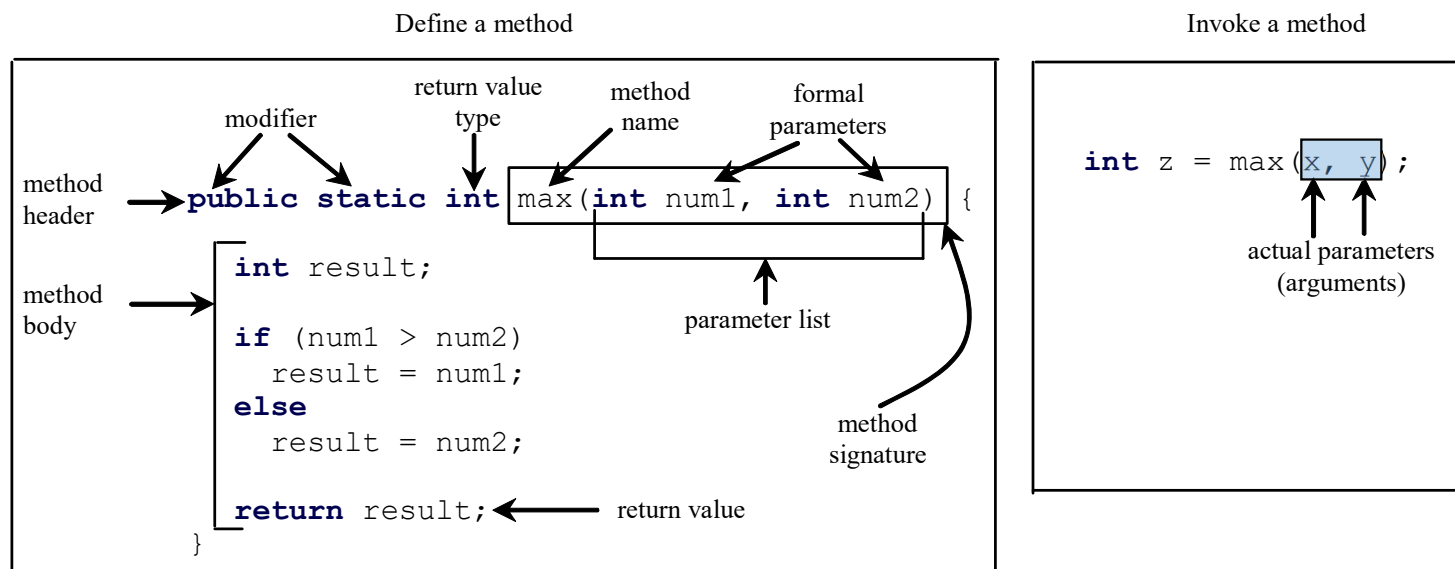
Formal Parameters

The variables defined in the method header are known as *formal parameters*.



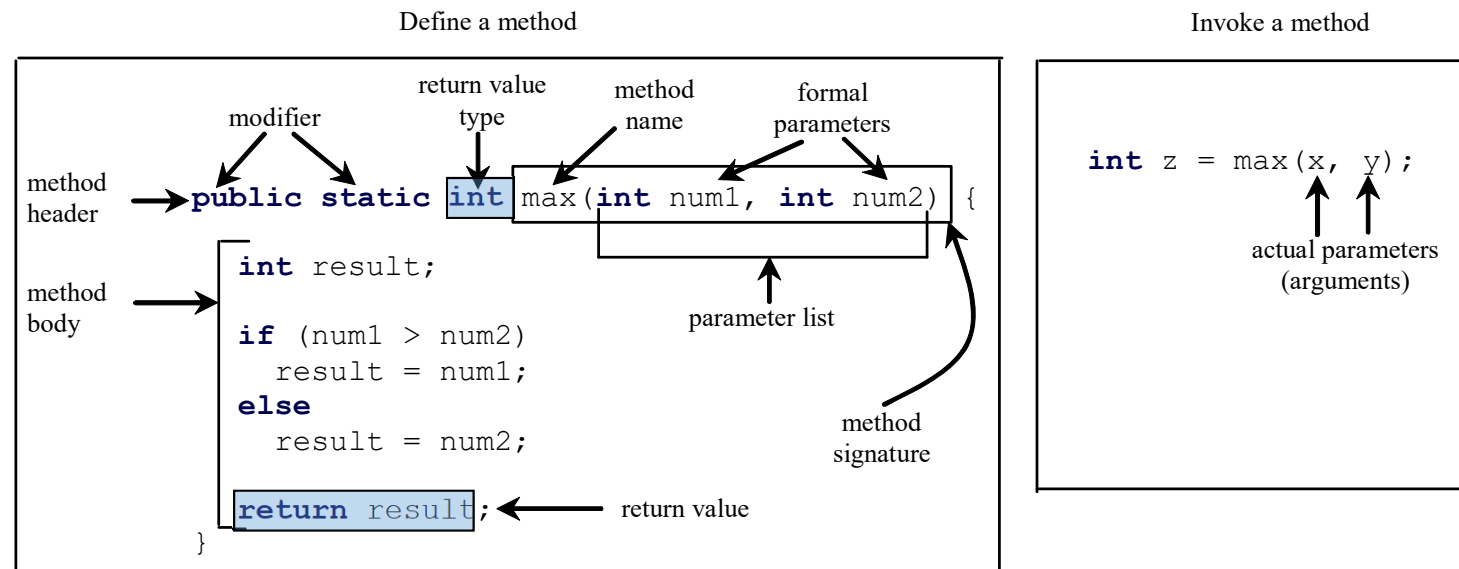
Actual Parameters

When a method is invoked, you pass a value to the parameter. This value is referred to as *actual parameter* or *argument*.



Return Value Type

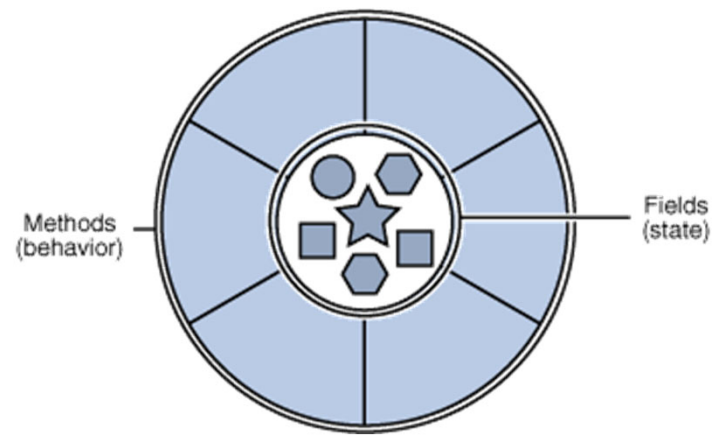
A method may return a value. The returnValueType is the data type of the value the method returns. If the method does not return a value, the returnValueType is the keyword void.



Objects

Object: An entity that contains data and behavior.

- *data*: variables inside the object
- *behavior*: methods inside the object
 - You interact with the methods; the data is hidden in the object.



Constructing (creating) an object:

Type **objectName** = new **Type** (**parameters**) ;

Calling an object's method:

objectName . **methodName** (**parameters**) ;

Bicycle: An example

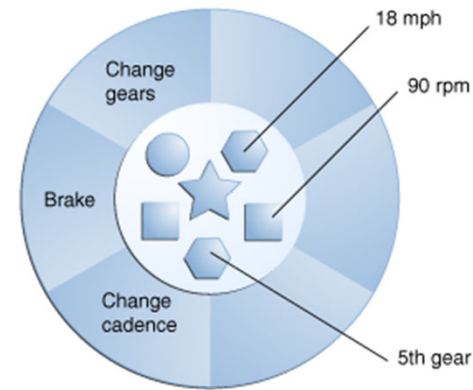
Software objects are similar to real world objects

They have *states* and *behaviors*

- *States* are stored in variables (Java: fields)
- *Behavior* is exposed via methods

Methods are used for object-to-object communication

Methods hide the internal state (and the concrete implementation) from the outside world (developer)

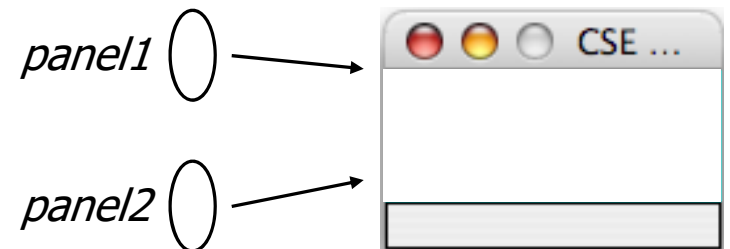


Object references

Arrays and objects use reference semantics.

- *efficiency*. Copying large objects slows down a program.
- *sharing*. It's useful to share an object's data among methods.

```
DrawingPanel panel1 = new DrawingPanel(80, 50);  
DrawingPanel panel2 = panel1;    // same window  
panel2.setBackground(Color.CYAN);
```



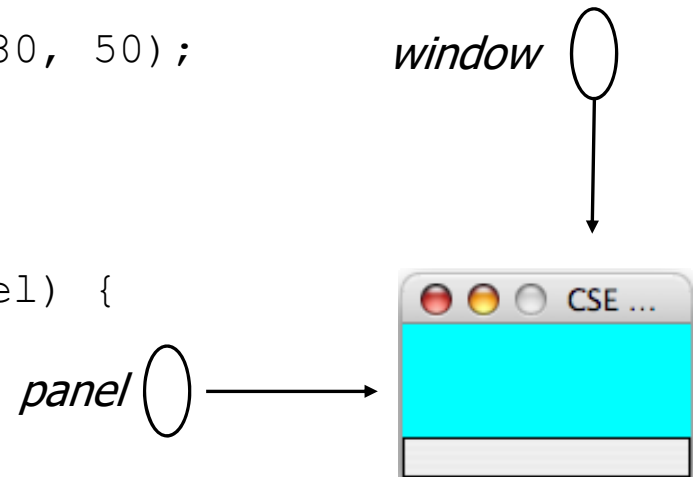
Pass by reference value (Objects)

When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.

- If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {  
    DrawingPanel window = new DrawingPanel(80, 50);  
    window.setBackground(Color.YELLOW);  
    example(window);  
}
```

```
public static void example(DrawingPanel panel) {  
    panel.setBackground(Color.CYAN);  
    ...  
}
```



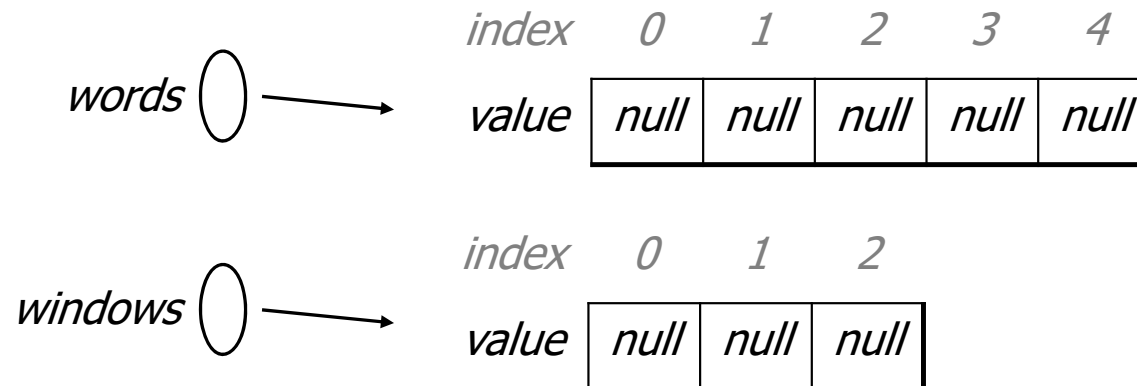
Arrays of objects

null : A value that does not refer to any object.

- The elements of an array of objects are initialized to `null`.

```
String[] words = new String[5];
```

```
DrawingPanel[] windows = new DrawingPanel[3];
```



Things you can do w/ null

store null in a variable or an array element

```
String s = null;  
words[2] = null;
```

print a null reference

```
System.out.println(s);      // null
```

ask whether a variable or array element is null

```
if (words[2] == null) { ...
```

pass null as a parameter to a method

```
System.out.println(null);   // null
```

return null from a method (often to indicate failure)

```
return null;
```

Null pointer exception

dereference: To access data or methods of an object with the dot notation, such as `s.length()`.

- It is **illegal** to dereference `null` (causes an exception).
- `null` is not any object, so it has no methods or data.

```
String[] words = new String[5];  
System.out.println("word is: " + words[0]);  
words[0] = words[0].toUpperCase(); // ERROR
```

Output:

```
word is: null
```

```
Exception in thread "main"  
java.lang.NullPointerException  
    at Example.main(Example.java:8)
```

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>
<i>value</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>

Static Variables, Constants, and Methods

Static variables are shared by all the instances of the class.

Static methods are *not* tied to a specific object.

Static constants are final variables shared by all the instances of the class.

CONCURRENCY ISSUES?

Instance Variables, and Methods

Instance variables belong to a specific class *instance*.

Instance methods are invoked by an *instance* of the class.

Class versus Instance Methods

Class methods are marked by the **static** keyword

Class methods are not called via an object reference but directly via the class

Can be called from a static context (e.g., main method)

Class methods often serve as utility function

They're generic and need no access to object variables and methods

```
class Datum {  
    private int day, month, year;  
    static String monthName(Datum d)  
    {  
        if (d.month == 1) return "January";  
        if ...           return "February";  
        ...  
    }  
    ...  
}
```

Inheritance (Briefly)

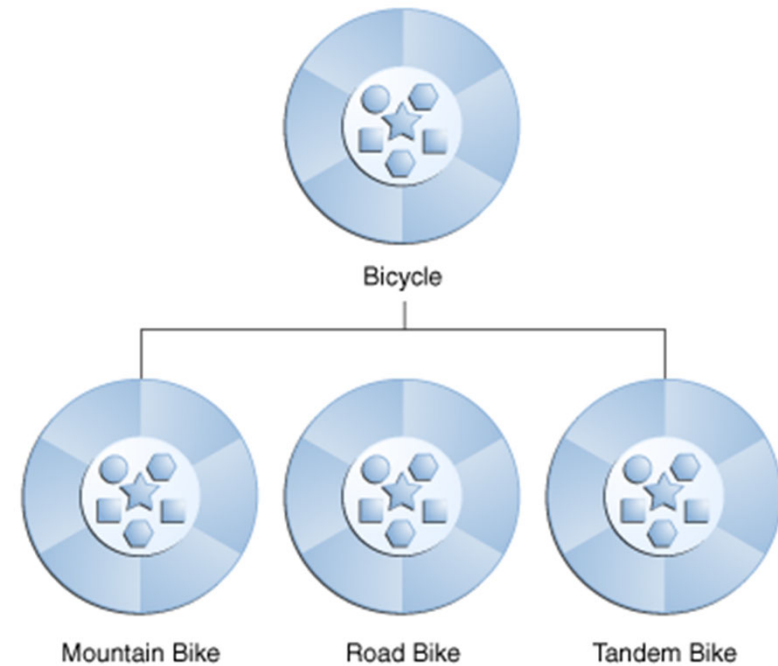
Different kinds of objects often have a certain amount in common with each other.

But there are systematic differences

Classes *inherit* commonly used state and behavior from other classes.

But specialize behavior and states further

```
class MountainBike extends Bicycle {  
  
    // new fields and methods defining  
    // a mountain bike would go here  
  
}
```



<http://docs.oracle.com/javase/tutorial/java/concepts/inheritance.html>

Defining a Subclass

A subclass inherits from a superclass. You can also:

- ➡ Add new properties
- ➡ Add new methods
- ➡ Override the methods of the superclass

Calling Superclass Methods

You could rewrite the printCircle() method in the Circle class as follows:

```
public void printCircle() {  
    System.out.println("The circle is created " +  
        super.getDateCreated() + " and the radius is " + radius);  
}
```

Overriding Methods in the Superclass

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

```
public class Circle extends SimpleGeometricObject {  
    // Other methods are omitted  
  
    /** Override the toString method defined in GeometricObject */  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
}
```

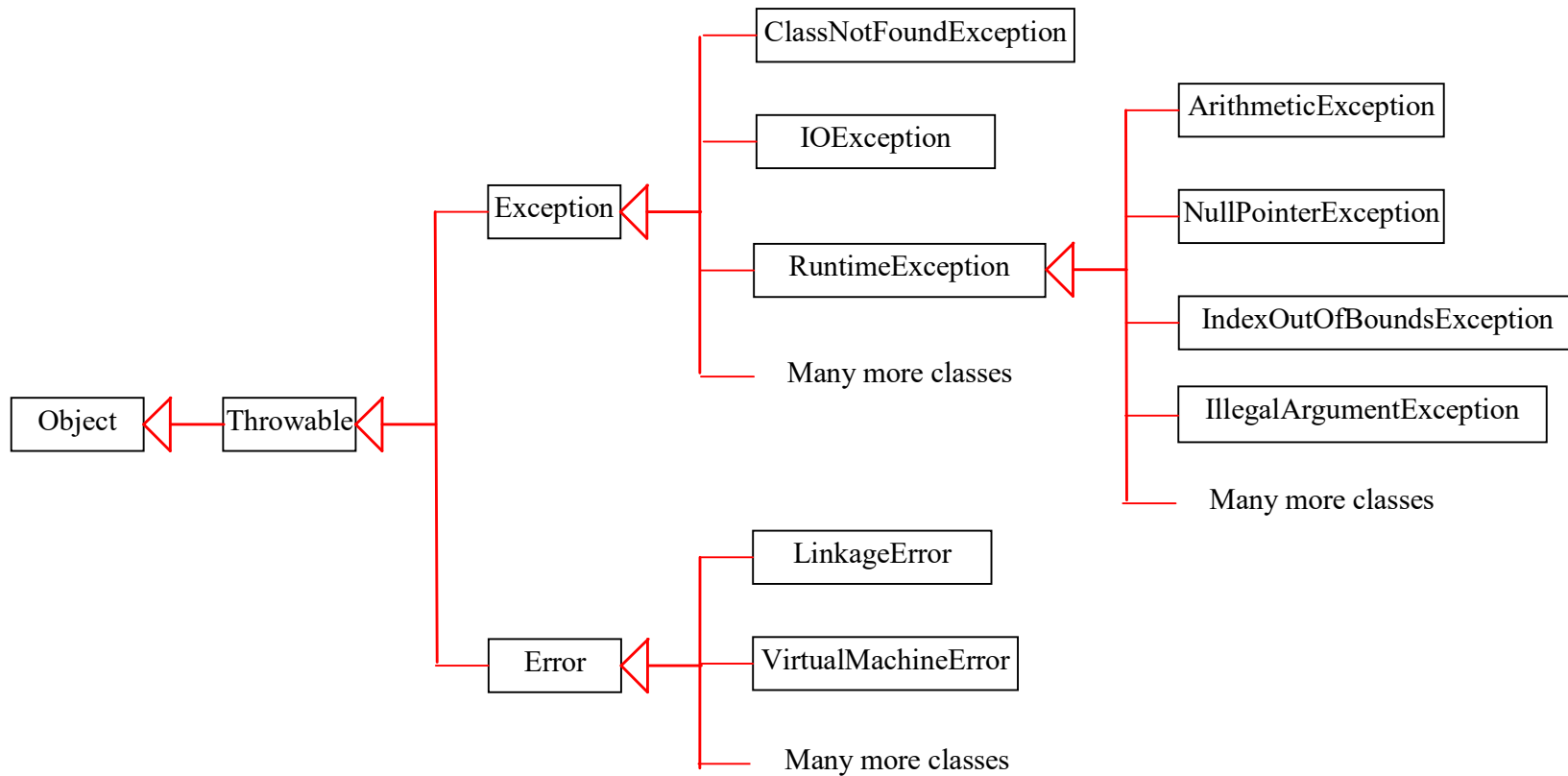
Exceptions

Advantages of exception handling:

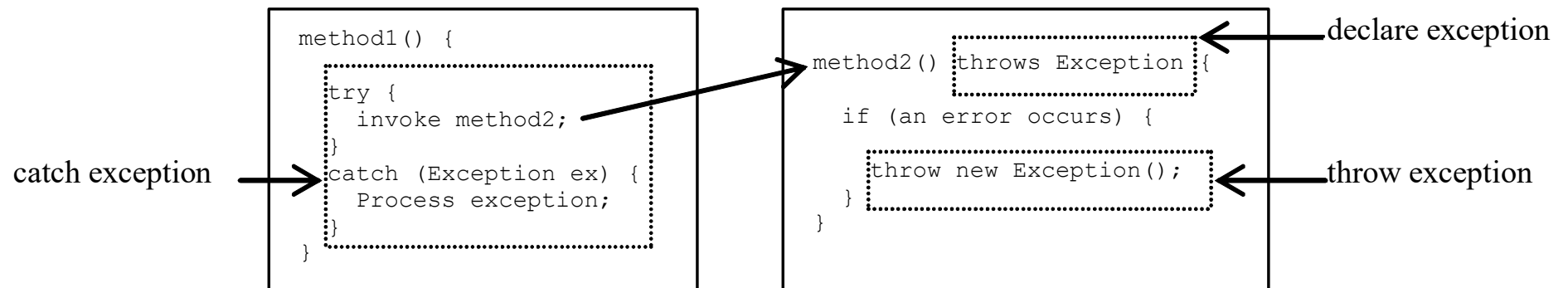
- enables a method to throw an exception to its caller
- without this, a method must handle the exception or terminate the program

```
public static int quotient(int number1, int number2) {  
    if (number2 == 0)  
        throw new ArithmeticException("Divisor cannot be zero");  
  
    return number1 / number2;  
}
```

Exception Types



Declaring, Throwing, and Catching Exceptions



Language features vs. parallelism: Guidelines

- Keep variables **as 'local' as possible**: global variables means they can be accessed by various parallel activities. While when its local to the process/thread, we are safe against inadvertent accesses to the variable.
- If possible, **avoid aliasing** of references: aliasing can lead to unexpected updates to memory through a process that accesses a seemingly unrelated variable (named differently).
- If possible, **avoid mutable state, in particular when aliased**: aliasing is no problem if the shared object is immutable, but concurrent mutations can make bugs *really* hard to reproduce and investigate (“Heisenbugs”)

Language features vs. parallelism: Guidelines

- Exceptions vs. Concurrency/Parallelism:
 - Exceptions tend to be less effective with parallelism because the **cause of the error** may be **far earlier** in the execution than where the exception triggers. Hence, the stack trace of the exception can be less informative and useful.
 - Exceptions thrown in a thread (parallel process) don't automatically reach the main program, and thus might go completely unnoticed. This can make it (even) more complicated to track down the root cause of a bug.

Summary

- Sneak peek into JVMs
 - Brief overview of JVM components ...
 - ... spending a bit more time on bytecode
 - This material is not examinable. Do not worry if you do not follow some of the concepts. This is only meant to give you a feel for the bigger picture (and can also be useful when debugging).
- Recap of Java
 - Certain constructs and patterns that are connected to concurrency
 - More recap slides in the slide deck, not covered in class