

Basic Java

Evgeny Borisov

bsevgeny@gmail.com

Who are you?

Big Data & Java Technical Leader

Mentoring

Consulting

Lecturing

Writing courses

Writing code

@jekaborisov

bsevgeny@gmail.com



Topics

- Java Basics:
 - Java Overview
 - Java Language
 - Data Types and Structures
 - Control Flow Statement
 - Object Oriented Programming - in Java
 - Classes and Objects
 - Inheritance and Polymorphism
 - Basic Java API Classes

Chapter one

Java Overview

Java Overview

1. What is Java?
2. History
3. Why Java?
4. Java Platform
5. Java Virtual Machine
6. Java API
7. The JDK
8. Java Platforms
9. Java Applications
10. CLASSPATH Environment Variable
11. The JDK Tools

History

- Developed by Sun Microsystems in 1991.
 - Research project to develop software for intelligent consumer electronics devices.
 - Designed with networking concepts in mind.
- Then used for creating Web pages with *dynamic content*.
- Now also used to:
 - Develop large-scale enterprise applications,
 - Enhance WWW server functionality,
 - Provide applications for consumer devices (cell phones, etc.).

Why Java?

- Java is Platform Independent.
 - Java programs can run on any system on which a Java Virtual Machine has been installed.
 - « ***Write once, run anywhere, reuse everywhere.*** »

Why Java?

- Java is a fully Object-Oriented language (almost) with strong support for proper software engineering techniques.
 - Everything is an object in Java (almost).
 - High quality architecture (almost).

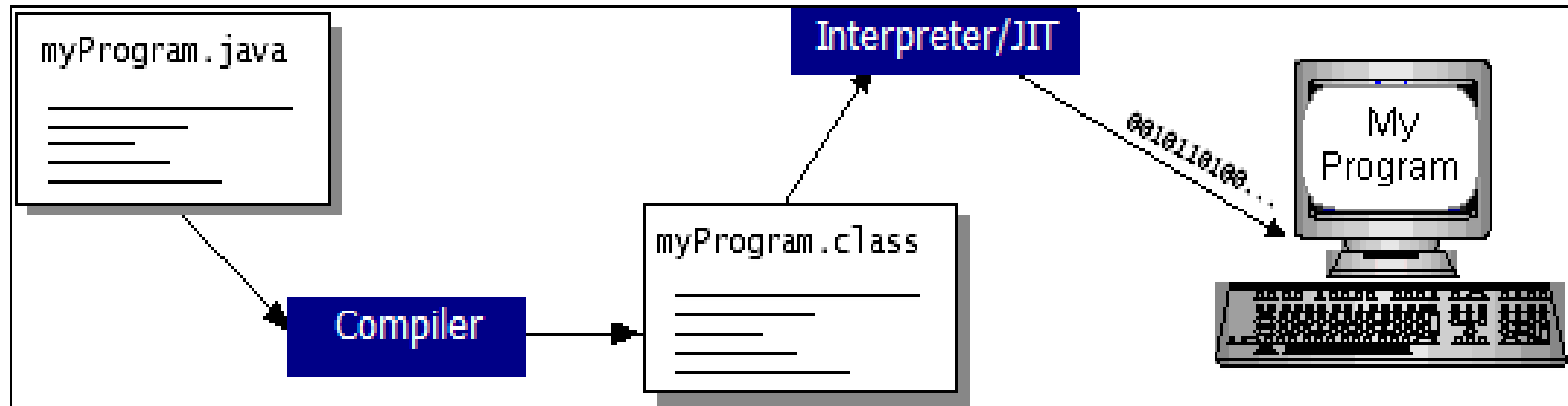
Why Java?

- Java is Easy to Learn and Easy to work with
 - Compared to C++
 - No pointers, memory management is automatic ...
- Error processing is built into the language:
 - Null pointer checking, Array-bounds checking, Exceptions, Bytecode verification, Automatic garbage collection.

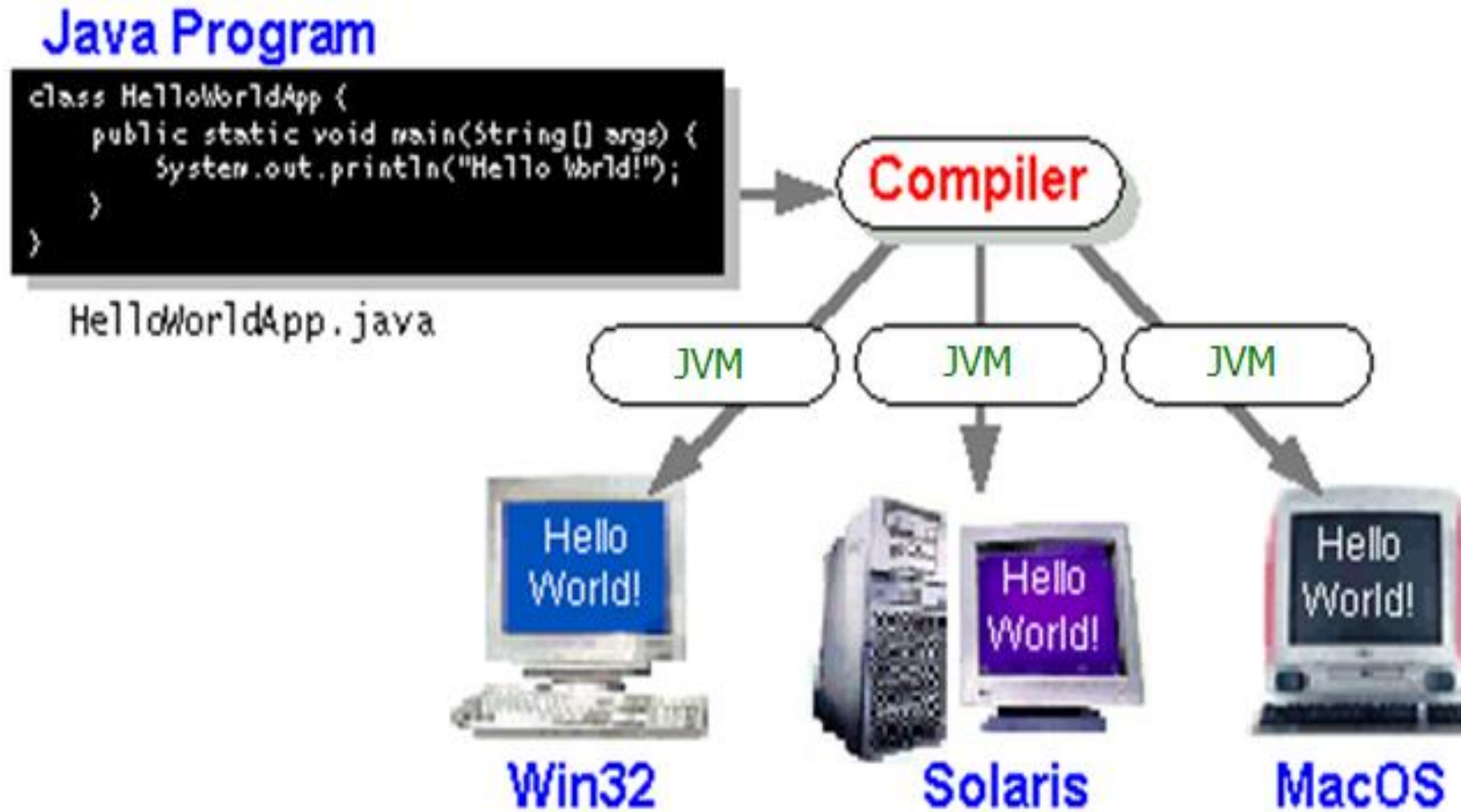
Why Java?

- Java is Secure:
 - The language "protects" :
 - The programmer (you can't write to OS)
 - Protect from users (person.setAge(-100))
- Java is Dynamic:
 - Linkage in runtime – ability to load updated libraries downloaded from the internet as the program is already running.
- Rich class library (Java API):
 - Graphics, IO, net, multimedia etc.

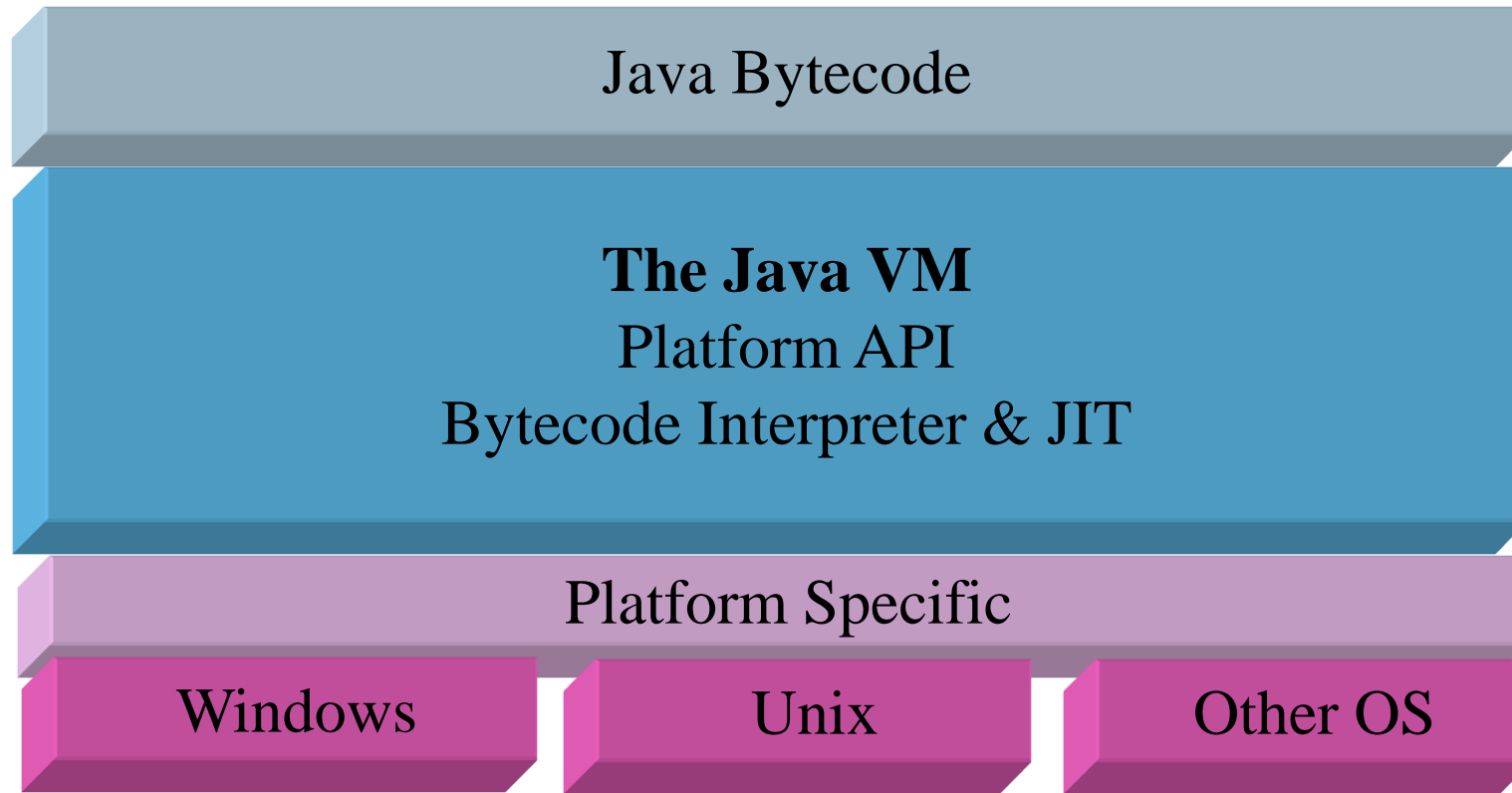
How java work



So than...



Something like this



Java API

- The Java API is a large collection of ready-made software components that provide many useful capabilities
 - Class library developed by Sun for use with the Java language
 - Designed to assist developers to write own classes and applications
 - Applications in Java can be written in few lines (leveraging libraries)
 - Isolation layer between the program and the platform

Java API

- The Java API is grouped into libraries of related classes and interfaces; these libraries are known as *packages*.
- Some of the features of the Java API:
 - The essentials:
 - Objects, strings, threads, numbers, I/O, data structures, system properties, date and time, networking, and so on.

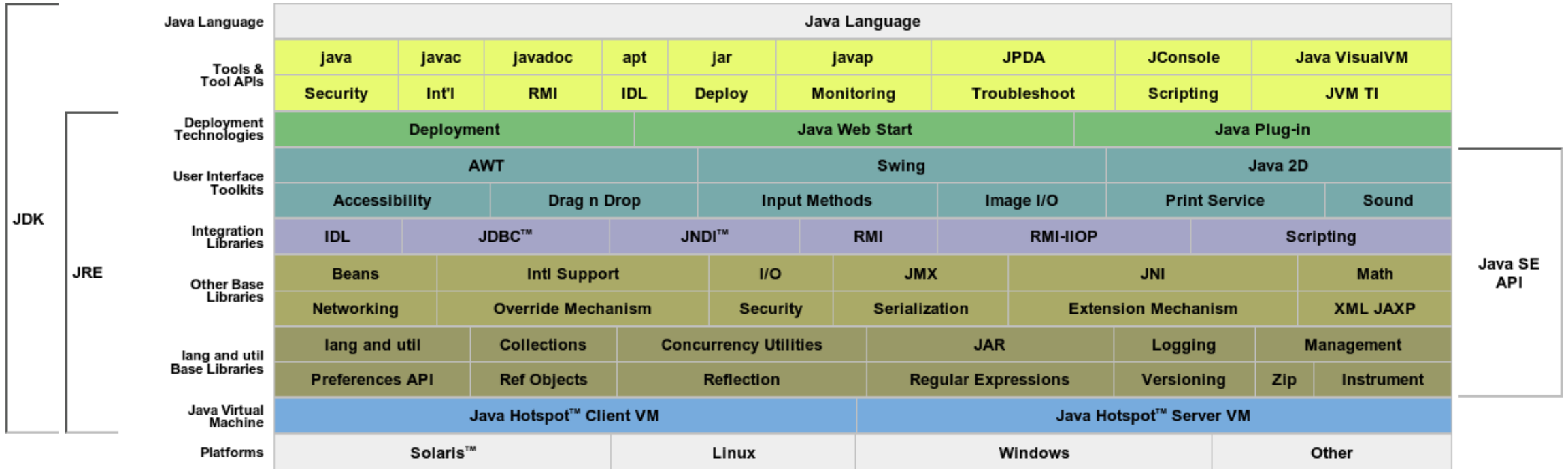
Java API

- Some of the features of the Java API:
 - Networking
 - Internationalization
 - Security:
 - Both low level and high level, including electronic signatures, public and private key management, access control, and certificates.
 - Java Database Connectivity (JDBC)
 - etc.

JDK

- JDK = Java Development Kit
 - Contents:
 - JRE - Java Runtime Environment
 - The virtual machine;
 - Java API;
 - Development Tools
 - Compiler, Debugger, Profiler...

JDK



Java Platforms

- Java Standard Edition (Java SE):
 - Simple applications, technical infrastructure work.
 - JDK + set of APIs such as : UI, I/O, networking...
- Java Enterprise Edition (Java EE):
 - Distributed applications.
 - Scalability, high availability
 - Other successful Java containers also exist (e.g. Spring)
- Java Micro Edition (Java ME)
 - Cellular phones, organizers, embedded...
 - Subset of Java SE + specialized libraries

Java platform languages

- Kotlin
- Groovy
- Scala
- JRuby
- Jython
- Another 30 😊

5 steps to run Java

1. Edit – Developer writes a code
2. Compile – byte is created by compiler
3. Load – Classloader load classes to heap/perm
4. Verify – verifier check that everything is ok (e.g. security)
5. Execute – jit will compile to native in runtime.

Java application types

- Applications – this course cover
- Applet – almost dead
- Container managed applications – next course 😊

Java Application

```
// Welcome1.java
// A first program in Java.
public class Welcome1 {

    // main method begins execution of Java application
    public static void main(String args[]) {
        System.out.println("Welcome to Java Programming!!!");
    } // end method main

} // end class Welcome1
```

Welcome1.java program output:

```
Welcome to Java Programming!
```

First app – lets start from comments

```
// Welcome1.java
```

- `//` one line comment (for IntelliJ: cntrl /)
- `/*` Block
of comments `*/` (for IntelliJ: cntrl shift /)

Class declaration

```
public class Welcome1 {
```

- Any class starts like this

Java conventions

- Class starts from UpperCase
- THIS_IS_CONSTANT
- Only here you can use _
- Never use this: _methodVariable=3
- Only class (type) starts from uppercase.
- Use upperCaseToDevideBewtweenWords

Java convention examples

- Interface name: `PersonService`
- Not `IPersonService` – this is not C#
- Class name: `PersonServiceImpl`
- Method name: `printPersonDetails`
- Variable names:
 `String name, int age, Person person`
- Constant: `final int NUMBER_OF_LEGS = 2`

Why use
conventions?



How many mistakes can you find here?



new balance()

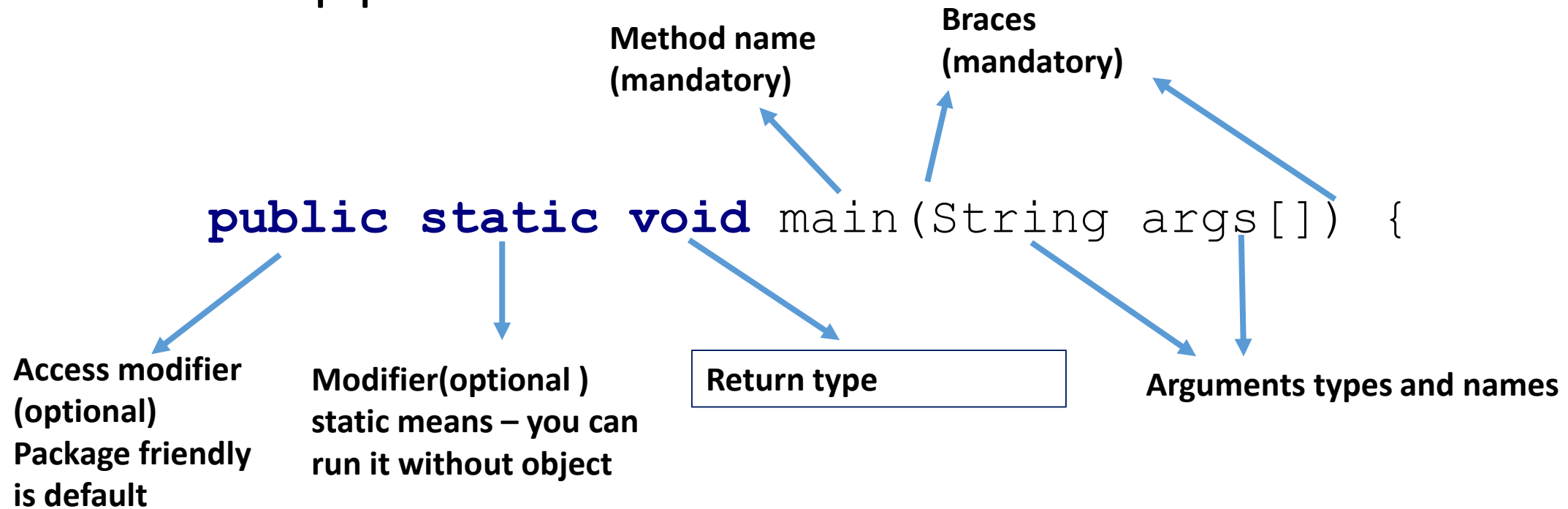
935 6549

NEW

NEW

NEW

First App



- You can run this method
- Main Thread will run it

How to run main?

Professional use only keyboard



How to run main?

- IntelliJ: cntrl + shift + F10 – run current main
- Shift + F10 – run last invoked main
- F9 for debug

Debug

- Cntrl + F8 – break point
- F9 – next break point
- F8 – step forward
- F7 – step into
Shift F8 – back
- Alt F8 – evaluate expression

First Program analyses – part of code

```
System.out.println("Welcome to Java Programming!");
```

- **System.out**
 - Standard output, in order to print to console
- Method **System.out.println**
 - Displays line of text;
 - Argument inside parenthesis;
 - Prints string of characters;
 - String - series characters inside double quotes. Text inside quotes can use variables or expressions. You can't do it: "x+y = \${x+y}" – It's not groovy or even C#
- This line known as a statement
 - Statements must end with semicolon (;)

Last slide of first program

```
} // end method main
```

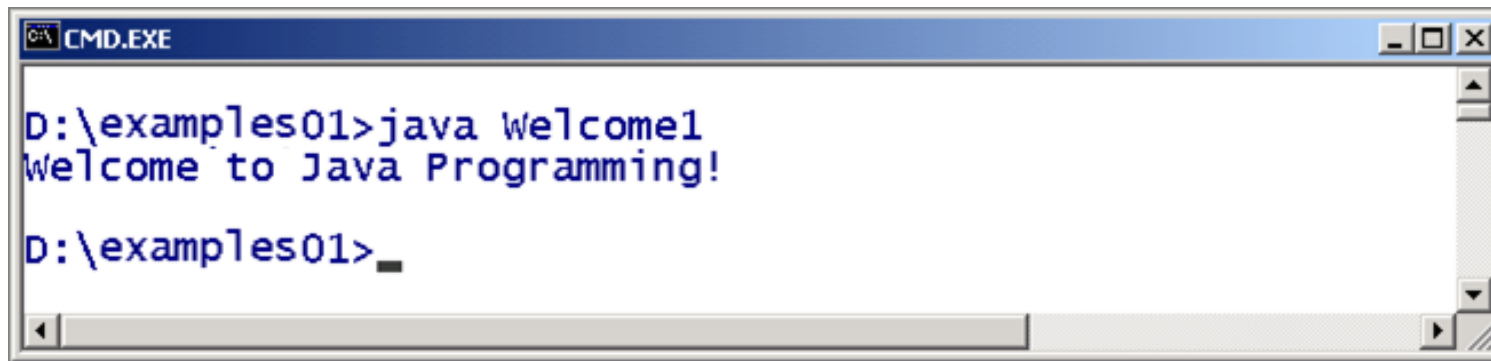
- Ends method definition

```
} // end class Welcome1
```

- Ends class definition

Compiling and running

- Write: `javac Welcome1.java` in command line, where this file exists
- If now compilations errors then `Welcome.class` will be created (bytecode of this class)
- Running



```
CMD.EXE
D:\examples01>java Welcome1
Welcome to Java Programming!
D:\examples01>_
```

The screenshot shows a Windows Command Prompt window titled "CMD.EXE". The prompt is at "D:\examples01>". The user has entered the command "java Welcome1", and the output "Welcome to Java Programming!" is displayed on the next line. The prompt is now at "D:\examples01>_".

The classpath environment variable

- Is part of the configuration process.
- Is used by the compiler and interpreter to find the java classes.
- Can be set:
 - Either by using compiler option -classpath
 - Or by setting the CLASSPATH environment variable.
 - `C:\>set CLASSPATH=.;/home/java/`

JDK contains

javac : the java compiler

java : the java interpreter

jar : the java archive tool

javah : the java code generator for calling C

javadoc : the java documentation generator

jdb : the java debugger

appletviewer : for viewing applets

javaws : for Java WebStart applications

JVisualVM

Various other development and monitoring tools

Javac compiler

- Usage : javac [option] *filename.java*
 - Option :
 - -classpath : specifies the classpath
 - -d directory: specifies root of the class hierarchy
 - -g : mode debug
 - -nowarn : to not print warning message
 - -nowrite : to not create any class files
 - useful to check that a file compiles
 - -verbose : display message about what the compiler is doing
- Creates a java byte-code suffix .class

The Java VM executable

- The *java* executable executes bytecode.
- Usage: *java fully-qualified-classname*
(if CLASSPATH variable is already set)

OR

- Usage: *java -cp classpath fully-qualified-classname*
 - Class name must explicitly include a package.
 - The suffix *.class* is omitted.

Chapter 2

Data Types and Structures

Index

1. Primitive Data Types.
2. Operators.
3. Arithmetic Operators.
4. Relational and Conditional Operators.
5. Logical Binary Operators (Boolean).
6. Shift Operators.
7. Arrays.

Primitive variable types

- int, double...
- Their size is platform independent (int always 4 bytes)
- All types should be declared (not like Java Script)

Primitive Data Types

- Arithmetic types:
 - byte
 - One byte (8-bit) signed two's complement integer.
 - short
 - 2 bytes (16-bit) signed two's complement integer.
 - int
 - 4 bytes (32-bit) signed two's complement integer.
 - long
 - 8 bytes (64-bit) signed two's complement integer.
 - float
 - 4-byte single-precision float.
 - double
 - 8-byte single-precision float.

Primitive Data Types

- Characters types
 - char
 - 2-byte unsigned Unicode character
- Boolean type
 - boolean
 - 1 bit boolean value (true/false)
 - 0/1 NOT WORKS!!!

Operators

- *unary operator* : requires one operand.
 - *operator operand*
 - *operand operator*
 - example `i++; ++i;`
- *binary operators* : requires two operands.
 - *operand1 operator operand2*
 - example `i + j;`
- *ternary operator* : requires three operands
 - *expression ? operand1 : operand2*
 - is short-hand if-else statement performed in-place.

Java's binary arithmetic operations:

<u>Operator</u>	<u>Use</u>	<u>Description</u>
+	op1 + op2	Adds op1 and op2.
-	op1 - op2	Subtracts op2 from op1.
*	op1 * op2	Multiplies op1 by op2.
/	op1 / op2	Divides op1 by op2.
%	op1 % op2	Computes the remainder of dividing op1 by op2.

Assignment Operators

Any statement of the form:

variable = variable operator expression;

Can be written as:

Variable operator= expression;

Assignment operator	Sample expression	Explanation	Assigns
<i>Assume: int c = 3, d = 5, e = 4, f = 6, g = 12;</i>			
+=	c += 7	c = c + 7	10 to c
-=	d -= 4	d = d - 4	1 to d
*=	e *= 5	e = e * 5	20 to e
/=	f /= 3	f = f / 3	2 to f
%=	g %= 9	g = g % 9	3 to g

Increment and decrement operators

Operator	Called	Sample expression	Explanation
++	preincrement	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postincrement	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	predecrement	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	postdecrement	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Relational operators

- Compares two values and returns the boolean logical value of this comparison (`true` or `false`) `>` , `>=` , `<` , `<=` , `==` , `!=`

Java equality or relational operator	Example of Java condition	Meaning of Java condition
<code>==</code>	<code>x == y</code>	<code>x</code> is equal to <code>y</code>
<code>!=</code>	<code>x != y</code>	<code>x</code> is not equal to <code>y</code>
<code>></code>	<code>x > y</code>	<code>x</code> is greater than <code>y</code>
<code><</code>	<code>x < y</code>	<code>x</code> is less than <code>y</code>
<code>>=</code>	<code>x >= y</code>	<code>x</code> is greater than or equal to <code>y</code>
<code><=</code>	<code>x <= y</code>	<code>x</code> is less than or equal to <code>y</code>

Logical operators.

&	executes a logical and, bit-by-bit.
	executes a logical or, bit-by-bit.
^	executes a logical xor, bit-by-bit.
~	executes a logical not, bit-by-bit.

Conditional & logical operators

What difference between & and &&, | and ||

- & and | are logical operators
- && and || are lazy operators

```
int i=1;
int x=1;
while (x++<10 && i++<10){
    // some stupid code
}
System.out.println(x);
System.out.println(i);
```

11
10

?

```
int i=1;
int x=1;
while (x++<10 & i++<10){
    // some stupid code
}
System.out.println(x);
System.out.println(i);
```

11
11

Conditional & logical operators

What difference between & and &&, | and ||

```
if (str != null & str.length() != 0) {  
    System.out.println("We love JAVA");  
}
```

?

Exception in thread "main" java.lang.NullPointerException

Conditional operators

- Conditional operators: five binary and one unary (only on boolean):

&& logical and, lazy evaluation.

logical or, lazy evaluation.

! logical not, unary operator.

Most important...

>> shift to the right, signed.

<< shift to the left.

>>> shift to the right, not(!) signed.

- All this operators was overridden in Groovy

Casting of primitives types



Examples

- `float f = 3.14` // Error. double is default
- `float f = 3.14f` // ok
- `float f = (float) 3.14` // ok, we did casting
- `long big = getHugeNumber()`
- `int x = (int) big`

Array: Allocation

- Dynamic allocation:

```
int tab[] = new int[3];  
String[] names = new String[8]
```

- Static initialization:

```
int[] tab = {5,6,12};  
String names[] = {"Lanister ", "Frodo", "Fistadantilus", "Feanor", " Scofield"}
```

- `Array.length` returns the array length.

- Read-only attribute.

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

- An array is destroyed when it is not referenced any more. Not when his elements are empty

Allocating an Array and Initializing Its Elements

// InitArray.java - Creating an array.

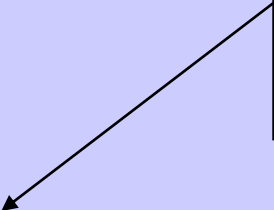
```
public class InitArray {  
  
    // main method begins execution of Java application  
    public static void main(String args[]) {  
        int array[];           // declare reference to an array  
        array = new int[10];  // dynamically allocate array  
        String output = "Subscript\tValue\n";  
  
        // append each array element's value to String output  
        for (int counter = 0; counter < array.length; counter++)  
            output += counter + "\t" + array[counter] + "\n";  
  
        System.out.println("Initializing Array of int Values");  
        System.out.println(output);  
    }  
}
```

Allocating an array and initializing its elements

Output:

```
Initializing Array of int Values
Subscript    Value
0            0
1            0
2            0
3            0
4            0
5            0
6            0
7            0
8            0
9            0
```

Each **int** is initialized
to **0** by default
null - in case of objects



Array index bounds

- When looping through an array:
 - Subscript should never go below 0
 - Subscript should be less than total number of array elements
- When an invalid array reference occurs:
 - Java generates `ArrayIndexOutOfBoundsException`
 - Exception handling is discussed further on.

Multiple-subscripted arrays

- Multiple-subscripted arrays:
 - Can be allocated dynamically.

- 3-by-4 array

```
int b[][];  
b = new int[ 3 ][ 4 ];
```

- Rows can have different number of columns.

```
int b[][];  
b = new int[ 2 ][ ]; // allocate rows  
b[ 0 ] = new int[ 5 ]; // allocate row 0  
b[ 1 ] = new int[ 3 ]; // allocate row 1
```

Multiple-subscripted arrays

- Static initialization:
 - Declaring and initializing double-subscripted (two-dimensional) array:

```
int b[] [] = { { 1, 2 }, { 3, 4 } };
```

- **1** and **2** initialize **b[0][0]** and **b[0][1]**
- **3** and **4** initialize **b[1][0]** and **b[1][1]**

```
int b[] [] = { { 1, 2 }, { 3, 4, 5 } };
```

- Row **0** contains elements **1** and **2**.
- Row **1** contains elements **3**, **4** and **5**.

Who need it today?



- Sometimes yes. (For instance Junit use it)

Chapter 3

Flow Control

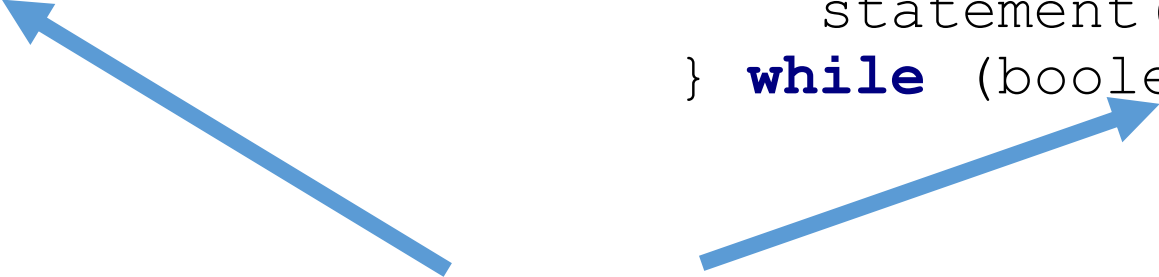
Index

- Overview
- Loops
 - while, do/while
 - for Statement
- Conditions
 - if-else
 - switch statements
- Break/Continue Statements

While

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

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



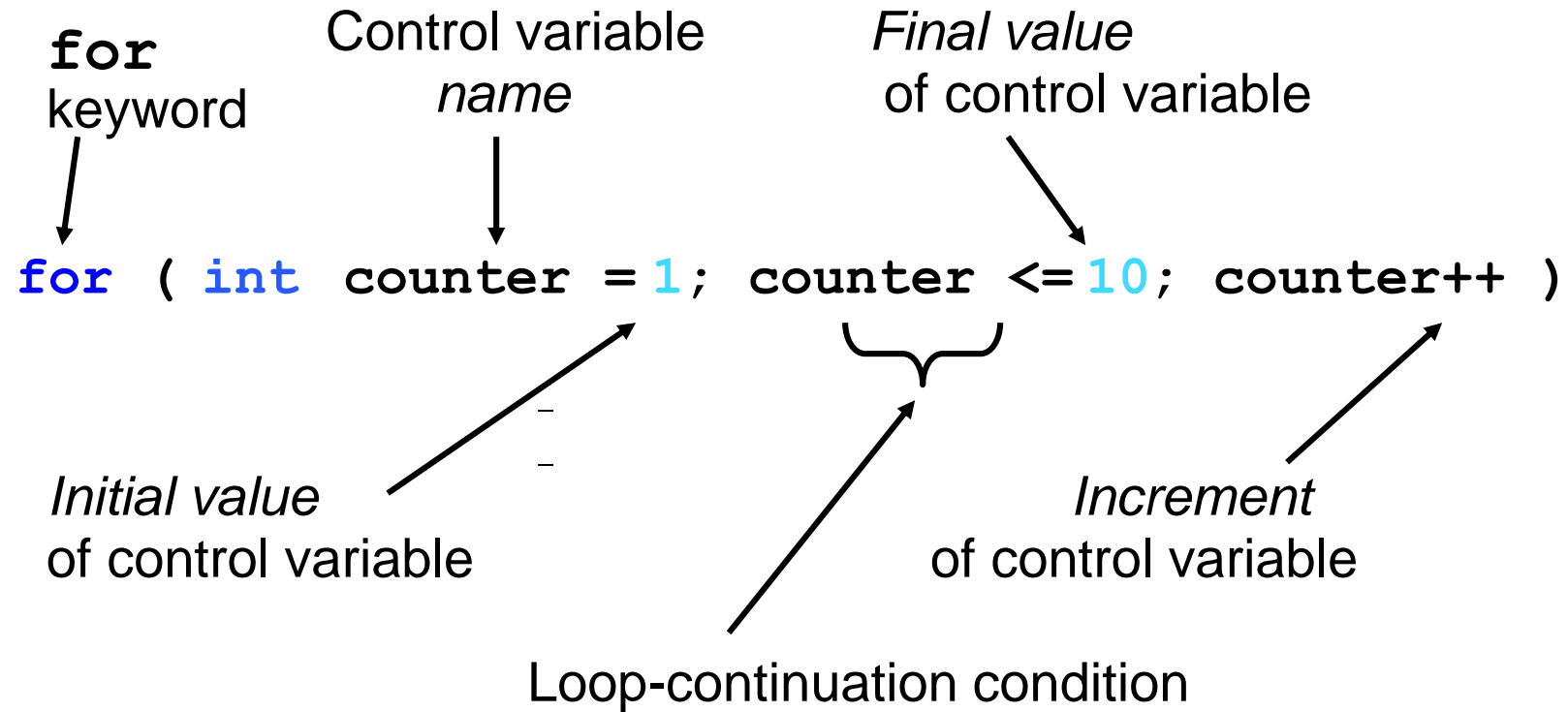
**Only boolean here.
It not c++ or groovy**

For

```
for (int i = 0; /* initialization */  
      i < 10; /* loop condition */  
      i++ /* iteration step */) {  
    statement(s);  
}
```

```
int[] arrayOfInts = {2, 4, 8, 16, 32, 64, 128, 256, 512, 1024};  
for (int i = 0; i < arrayOfInts.length; i++) {  
    System.out.print(arrayOfInts[i] + " ");  
}
```

For



In inteliJ type itar and press tab

Since Java 6

```
for (Value variable : collection) {  
    statement(s);  
}
```

```
int[] arrayOfInts = {1,2,3};  
for (int i : arrayOfInts) {  
    System.out.print(i + " ");  
}
```

- It was called For Each... 😊
- But real `forEach` exists only since Java 8
- This one implemented with iterator
- In IntelliJ type `iter` and press tab


Decision-making if-else statement

- The most basic `if` single statement:

Only boolean here.

It not c++ or groovy

```
if (booleanExpression) {  
    statement(s);  
}
```



- The statement `if` with a companion `else` statement:

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

- Conditional operator (`? :`)

```
int a = 3;  
int b = 5;  
System.out.println("max = " + (a > b ? a : b));
```


Switch...

```
switch (expression) {  
    case value1:  
        statement(s);  
        break;  
    case value2:  
        statement(s);  
        break;  
    // ...  
    default:  
        statement(s);  
        break;  
}
```

- Can work with integers, chars and enums
- Strings since Java 7
- Anti-pattern in Java

Statements **break** and **continue**

- Alter flow of control:
 - **break** statement -
 - Causes immediate exit from control structure
 - Used in `while`, `for`, `do/while` or `switch` statements.
 - **continue** statement
 - Skips remaining statements in loop body.
 - Proceeds to next iteration.
 - Used in `while`, `for` or `do/while` statements.

Lets write some code...

- Declare array of doubles and fill it
- Declare array of ints
- All round numbers must be copied to array if ints

Chapter 4

Objects and Classes

Agenda

- Class declaration and body
- Member Variables declaration
- Access Levels
- static, final
- Method declaration and body
- Constructors (Creating and using objects)
- References (memory concept)
- Passing parameters (by value & By reference)
- Argument Coercion
- Duration of Identifiers (scope rules)
- Method Overloading

package name

```
package demo;
```

Import for all classes of package java.util in order to use them in this class

```
import java.util.*;
```

Attributes. Part of an object. Stored with it on the heap memory

```
public class Car {  
    private final int id;  
    private int maxSpeed = 220;
```

Constructor. Invoked once object created

```
    public Car(int id) { this.id = id; }
```

Methods

```
    public void drive() { ... }
```

Methods (setters & getters)

```
    public int getMaxSpeed() {  
        return maxSpeed;  
    }
```

```
    public void setMaxSpeed(int maxSpeed) {  
        this.maxSpeed = maxSpeed;  
    }
```

```
}
```

Static Import

- Import class static members to save us reading them over and over again:
 - `Math.cos`
 - `Collections.min`
- `*` wildcard imports all static members in the class

```
import static java.lang.Math.cos;
```

```
// ...
```

```
double x = r * cos(theta);
```

Access Modifiers

	class	subclass	package	world
private	X			
protected	X	X	X	
public	X	X	X	X
package	X		X	

The Class Declaration

```
[Access Level | abstract | final] class <ClassName>  
    [extends <SuperClassName>] [implements  
    <InterfaceName>] {  
    classbody  
}
```

- The required components are the **class** keyword and the **class-name**
- Class names start with an uppercase letter, as a convention.

The Class Body

- The class body contains :
 - Constructors for initializing new objects,
 - Declarations for the attributes/variables that define the state of the class and its objects.
 - Methods to implement the behavior of the class and its instance objects.
 - Finalize method for cleaning up object's resources after it has done its job (rarely used).
- Variables and methods are collectively called "class members"

The Class Body – what else can be in it?

- Static blocks – invoked by classloaders when class will be loaded
- Initializers – anonymous blocks. Invoked before constructor once object created

```
public class SomeClass {  
    static {  
        System.out.println("I'm static block");  
        System.out.println("I will be invoked only once");  
    }  
    {  
        System.out.println("I'm initializer");  
        System.out.println("I will be invoked every time, once object created");  
    }  
}
```

Object variables (attributes)

- An object stores its state in variables.
- A variable has a name and a type
- A variable declaration is the association of the variable name with the type.

[Access-Level | static | final | transient...] *type name*

- Example:

```
private int maxSpeed;
```

- Variable names start with a lowercase letter, as a convention.

Object variables (attributes)

- You can apply them via comma: `person.name = "Lambert"`
- But Java conventions is declare fields as private
- So you apply them via setters/getters
- In order to generate them in IntelliJ press alt + insert
- `person.setName("Ned Stark")`
- `System.out.println(person.getName())`

Variables – default values

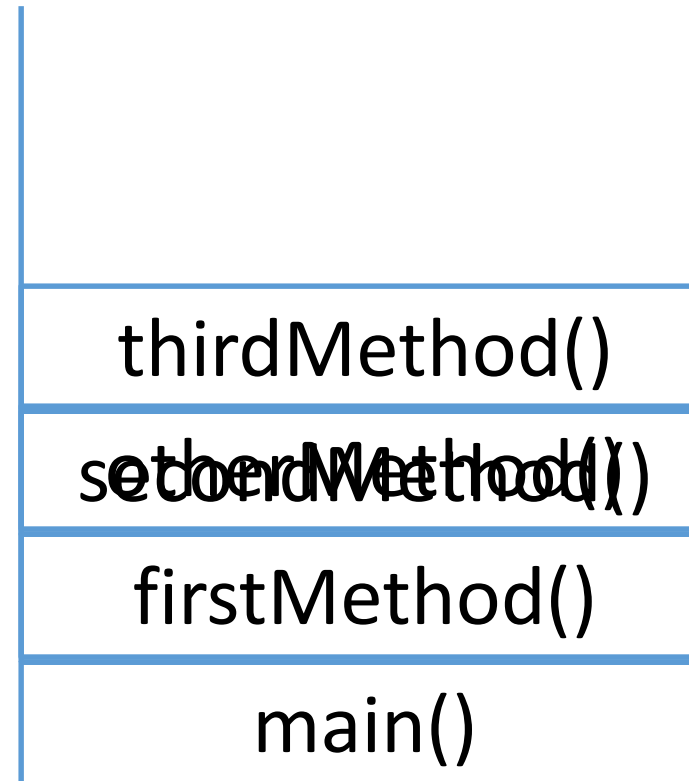
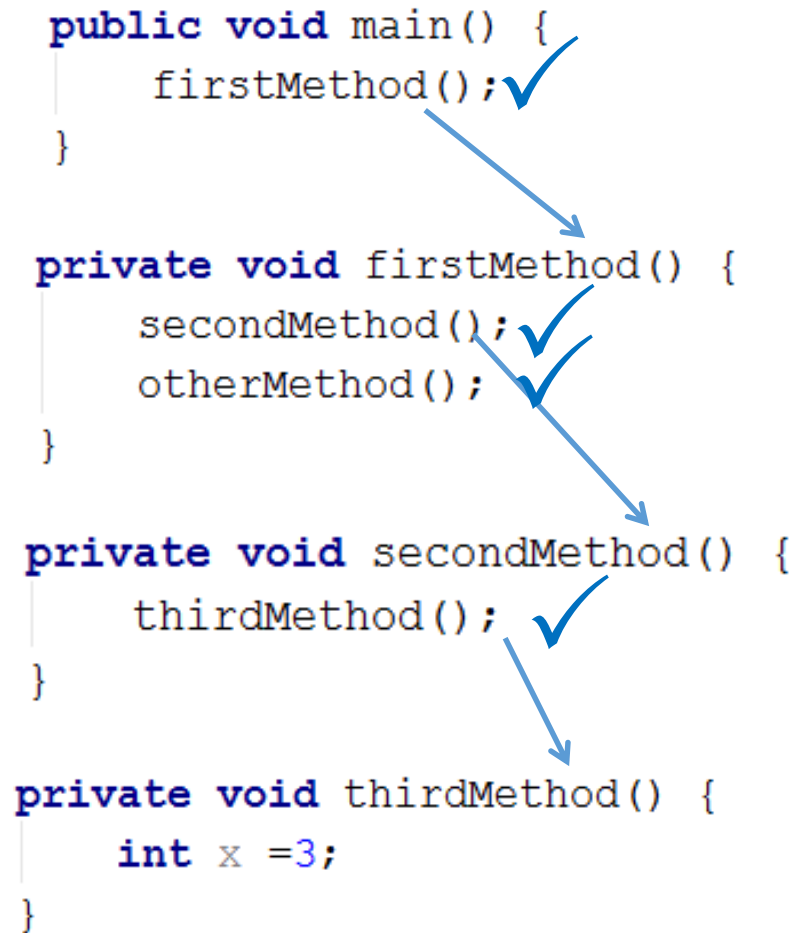
- Local variables (variables of some inner block, like method, constructor...) don't have default values. They are not part of an object. They exist on stack memory and live till the block where they are declared is working.

```
public void scream(boolean flag) {  
    int tagil;  
    System.out.println(flag);  
    if (flag) {  
        System.out.println(tagil);  
    }  
}
```

Variable 'tagil' might not have been initialized

Stack memory

```
public void main() {  
    firstMethod();  
}  
  
private void firstMethod() {  
    secondMethod();  
    otherMethod();  
}  
  
private void secondMethod() {  
    thirdMethod();  
}  
  
private void thirdMethod() {  
    int x = 3;  
}
```



Variables – default values

- Variables which are state of the object will get a default values when object will be created.
 - Numbers – 0
 - boolean - false
 - Reference – null
- You can override defaults values via constructor or via inlines



Inline

- Variables can be initialized when they are declared
- The data type of the variable must match the data type of the value assigned to it

```
private char myChar = 'S';  
private boolean flag = true;  
private int num = 3;
```

Static variable

- Is not part of an object, but class metadata
- Stored in permanent memory (Java 7 and down)
- Declaration: `static int counter`
- You can apply for them via object, but it is stupid. Correct way to apply them is via class: `Person.counter++`
- `static double perimeter = Math.PI * 2 * radius;`

Final Variable

- Example: `final int NUMBER_OF_DAYS_IN_A_WEEK = 7;`
- Constants names in uppercase, with underscores between words as a convention.
- If the value won't change, use 'final'. 'final' is your friend!
 - good programming
 - helps JIT optimizations

Method Declaration

```
[abstract | static | Access Modifier | final] returntype  
methodName (parmlist) [throws exceptions] {  
    method body  
}
```

- Example:

```
void setMaxSpeed(int maxSpeed) {  
    this.maxSpeed = maxSpeed;  
}
```
- Method names start with a lowercase letter, as a convention.

Method Declaration

- **abstract**
 - Method has no implementation
 - must be a member of an abstract class
- **static**
 - Declares this method as a class method rather than an instance method
- **Access Modifiers**
 - `public\protected\package\private`
 - If not specified - default is package access.
- **final**
 - A final method cannot be overridden by subclasses

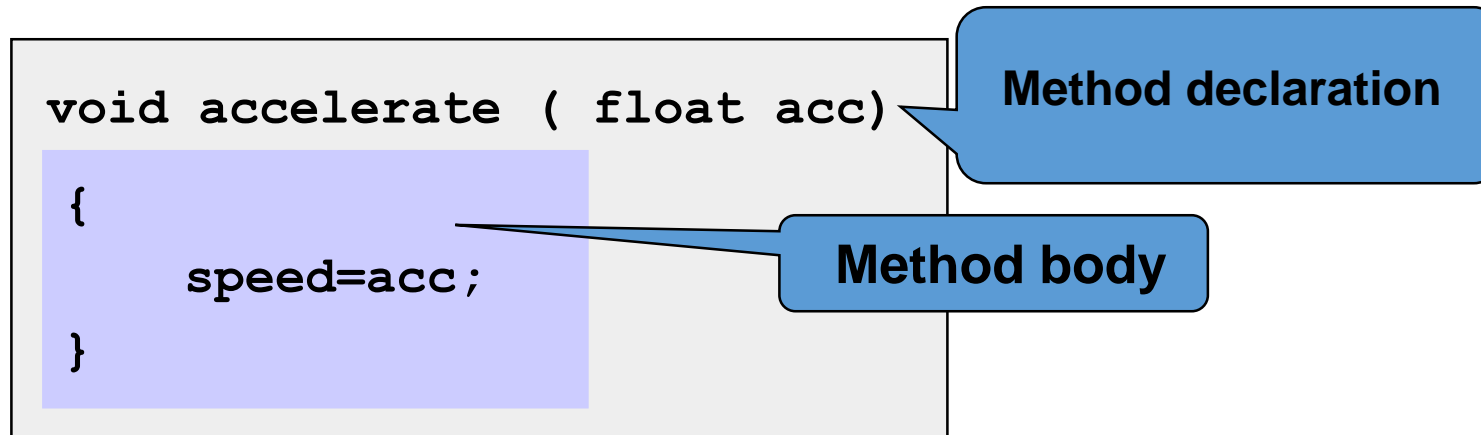
Method Declaration

- varargs (Variable arguments)
 - Method can take multiple arguments of the same type
 - Number of arguments not pre-determined (at compile- or runtime)
 - Marked with ellipsis (...) in param list
 - Access varargs as a regular array
 - No varargs supplied – empty array

```
private String[] features;  
public void setFeatures(String... features) {  
    this.features = features;  
}
```

Implementing Methods

- Like a class, a method has two major parts:
 - method declaration (signature)
 - The method declaration defines the method's access level, return type, name, and arguments
 - method body
 - Where all the action takes place. It contains the Java instructions that implement the method.



The Method Body: this

- `this` is a self-referencing mechanism.
- Within an object's method body you can refer the object's member variables and methods by using `this` keyword.

Constructors

- Same name as class
- Initializes instance variables of a class object
- Called when program instantiates an object of that class
- Can take arguments, but *cannot return data types*
- Class can have several constructors, through *overloading*

Constructors

- Example:

```
class Rectangle {  
    int width, height;  
    int x, y;  
  
    Rectangle() { /*...*/ }  
    Rectangle(int width, int height) { /*...*/ }  
    Rectangle(int x, int y, int width,  
              int height) { /*...*/ }  
    /*...*/  
}
```



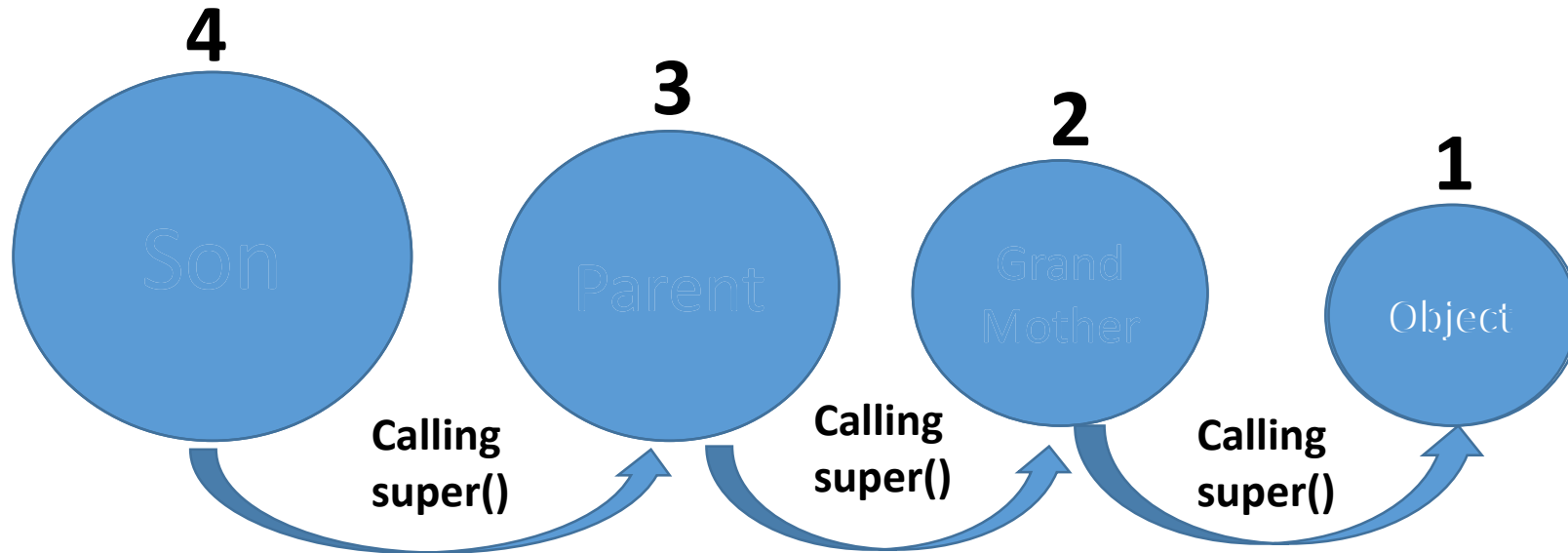
```
Rectangle rectangle = new Rectangle(5, 6);
```

Constructors

- If no constructor is defined - a default public empty constructor without arguments is supplied by the compiler
- If any constructor *is* defined, however – the compiler will not define a default constructor
- Constructors are not inherited!

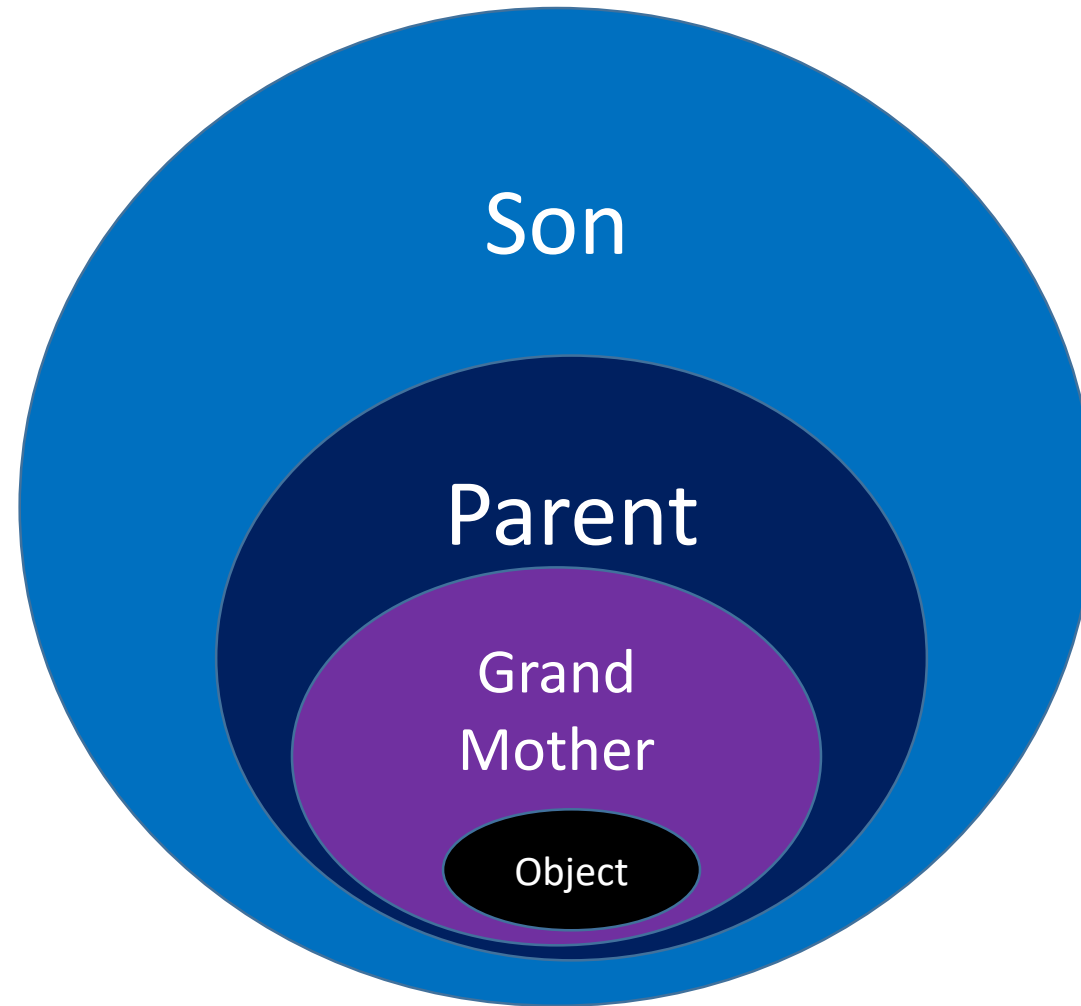
Constructor chaining

What happen when new Son object is created?



Yes. Object construction is recursive process

Constructor chaining



Constructors

- What would happen here?

```
class BaseClass {  
    BaseClass(int x)  { /*...*/}  
    BaseClass(String y) { /*...*/}  
}
```

```
class DerivedClass  
    extends BaseClass { /*...*/}
```

Can be fixed in 2 ways

- Find by yourself
- You can ask IntelliJ (alt enter)


Riddle

```
public class Parent {  
    private int x;
```

```
    public Parent(int x) {  
        this.x = x;  
    }
```

```
    public int getX() {  
        return x;  
    }
```

```
}
```



```
public class Son extends Parent {
```

There is no default constructor available in 'course.Parent'

```
    public void goodMethod(){  
        //good code here  
    }
```

```
}
```

Any problems?

Riddle

```
public class Parent {  
    private int x;
```

```
    public Parent() {  
    }
```

```
    public Parent(int x) {  
        this.x = x;  
    }
```

```
    public int getX() {  
        return x;  
    }
```

```
}
```

```
public class Son extends Parent {  
    private int x;
```

```
    public void goodMethod(){  
  
    }
```

```
    public void setX(int x) {  
        this.x = x;  
    }
```

```
}
```

What if I want to change x in Son class?

Constructor

```
public class Son extends Parent {  
    private int x;
```

```
    public void goodMethod(){  
  
    }
```

```
    public void setX(int x) {  
        this.x = x;  
    }  
}
```

```
public static void main(String[] args) {  
    Son son = new Son();  
    System.out.println(son.getX());  
    son.setX(3);  
    System.out.println(son.getX());  
}
```

```
public class Parent {  
    private int x;
```

```
    public Parent() {  
    }
```

```
    public Parent(int x) {  
        this.x = x;  
    }
```

```
    public int getX() {  
        return x;  
    }
```

```
}
```

? 00

Constructor

```
public class Son extends Parent {  
    private int x;  
  
    public void goodMethod(){  
  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
}
```

Solution?

```
public class Parent {  
    private int x;  
  
    public Parent() {  
  
    }  
    public Parent(int x) {  
        this.x = x;  
    }  
    public int getX() {  
        return x;  
    }  
}
```

Override getter

Creating an Object

- In Java, an object is created by creating an instance of a class
- **new operator** `Person person = new Person("Belov Pavel");`
- This single statement has three implications:
 - Declaration - variable person declared
 - Instantiation – Person object created
 - Initialization – Person object assigned to person variable (address of this object)

Using an Object

- When the object has been created you may
 - Need information from it,
 - Want to change its state, or
 - Have it perform some action.
- Two ways to do this:
 - Manipulate or inspect its variables.
 - Call its methods.

Using an Object

- The notation used to call an object's methods:
 - `objectReference.methodName (argumentList) ;`
 - Example:
`susita.accelerate (100) ;`

Memory Concepts

- Every variable has a name, a type, a size and a value
- The variable name *refers* to the data that the variable contains.
 - Name corresponds to location in memory

Memory Concepts

```
MyClass myObject;
```

- The above line defines 'myObject' as a reference type to an object of type MyClass
 - at this point 'myObject' doesn't contain any real object
 - it is initialized to **null** by the system (in case it is not local variable, remember?)

Reference

- A reference is a pointer set on an object
 - dynamically by using `new`
 - `this` : reference on the current object
 - Any variable containing a reference can be set to `null`
 - a reference on object can be compared to `null`

```
if (twingo != null)
    System.out.println("the twingo car exists");
```

Reference

- Array of Objects:
 - is actually array of references:

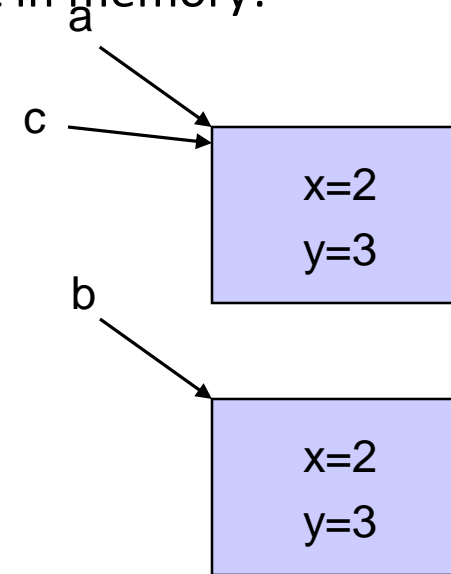
```
// first allocate the array
Spam mySpamArray[] = new Spam[10];

// now allocate an object for each cell
// in the array
for (int i=0; i<mySpamArray.length; i++) {
    mySpamArray[i] = new Spam();
}
```

Reference Operators

- Comparison operator on references
 - `==` checks if two references are equal
 - `!=` checks if two references are different
 - `a == b` only if physically they refer to the same object in memory!

```
Point a = new Point(2,3);  
Point b = new Point(2,3);  
  
if (a==b) { /*...*/ } // false!  
  
Point c = a;  
b=c;  
  
if (b==a) { /*...*/ } // true
```



Reference Operators

- `instanceof`: checks whether an object referred to is an instance of a given class

```
if ( susita instanceof Car )
```

Passing parameters in JAVA

- By value or by reference?
- Primitive?
- Object?

Passing variables

```
public class IntegerMutator {  
  
    public void doSomethingGoodWithX(int x) {  
        x++;  
    }  
}  
  
public static void main(String[] args) {  
    int x = 13;  
    IntegerMutator integerMutator = new IntegerMutator();  
    integerMutator.doSomethingGoodWithX(x);  
    System.out.println(x);  
}
```

?

13

Passing objects

```
public class ClientService {  
  
    public void tapelTapel(IDIClient client){  
        client.setDebt(1000);  
    }  
}  
  
public static void main(String[] args) {  
    IDIClient client = new IDIClient();  
    ClientService clientService = new ClientService();  
    clientService.tapelTapel(client);  
    System.out.println(client.getDebt());  
}
```

?

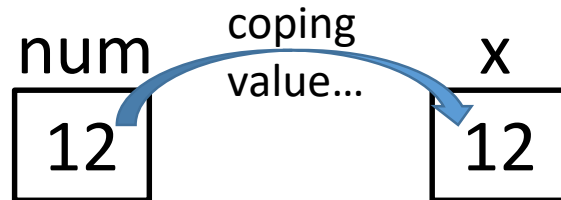
1000

Method arguments

- primitive type pass by value

```
public int doSomethingGoodWithX(int x){...}
```

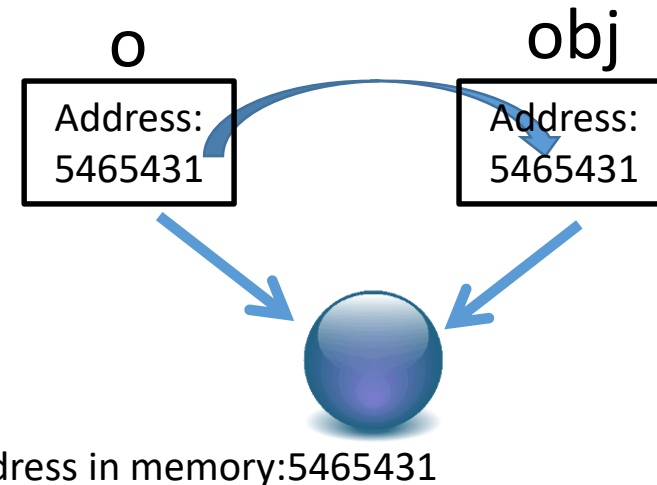
```
int num=12;  
doSomethingGoodWithX(num);
```



- object type pass by value of reference

```
public void tapelTapel(Object obj){...}
```

```
Object o = new Object();  
tapelTapel(o);
```



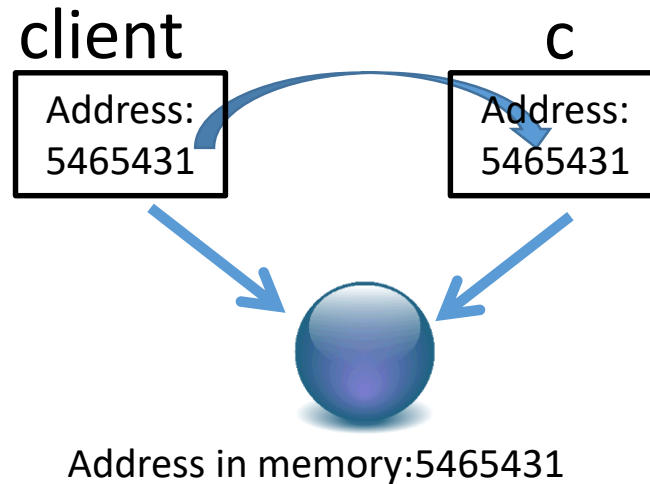
Method arguments

- what about immutable object?

```
IDIClient client = new IDIClient.Builder().name("Vadim").createIDIClient();
ClientService clientService = new ClientService();
clientService.tapelTapel(client);
System.out.println(client.getDebt());

public void tapelTapel(IDIClient client){
    client.debt(1000);
}
```

? 0



Waiting for GC



What about arrays?

```
public class ArrayService {  
    public void tapelTapelArray(int[] nums){  
        for (int i=0;i<nums.length;i++) {  
            nums[i]++;  
        }  
    }  
}  
  
public static void main(String[] args) {  
    int[] nums = new int[5];  
    ArrayService arrayService = new ArrayService();  
    arrayService.tapelTapelArray(nums);  
    for (int num : nums) {  
        System.out.println(num);  
    }  
}
```

?

1 1 1 1 1

Argument Coercion

- Coercion of arguments
 - Forcing arguments to appropriate type to pass to method
 - Example:

```
Math.sqrt ( 4 ) ;
```



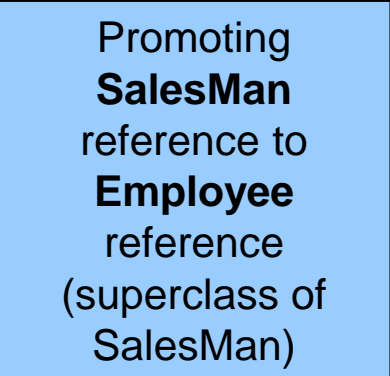
•Promoting
int to
double

The diagram consists of a blue rectangular box with a black border. Inside the box, the text '•Promoting' is on the first line, 'int' is on the second line, and 'double' is on the third line. An arrow originates from the top-right corner of the box and points diagonally upwards and to the left, ending at the space between the opening parenthesis and the number 4 in the code 'Math.sqrt (4) ;'.

Argument Coercion

- Another Valid promotion:
 - Subclass reference to Superclass reference
- Example:

```
SalesMan salesMan = new SalesMan();  
office.addNewEmployee( salesMan );
```

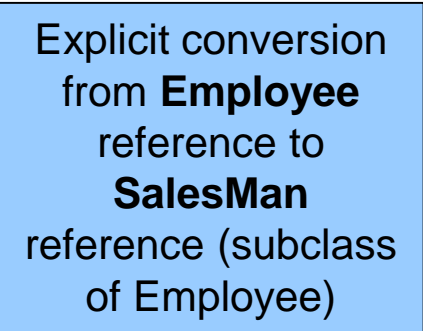


Promoting
SalesMan
reference to
Employee
reference
(superclass of
SalesMan)

Argument Coercion

- Converting variables in the other direction of the allowed promotion may result in:
 - loss of data, for primitives
 - weaker design, for objects
 - programmer can use cast operator to explicitly force the conversion
 - should be used very carefully
 - Example:

```
void addNewEmployee(Employee newEmployee) {  
    // ...  
    if (newEmployee instanceof SalesPerson) {  
        SalesPerson newSalesPerson = (SalesPerson)newEmployee;  
        // call methods specific to SalesPerson  
        newSalesPerson.setCommissionRate(/*...*/);  
    }  
    // ...  
}
```

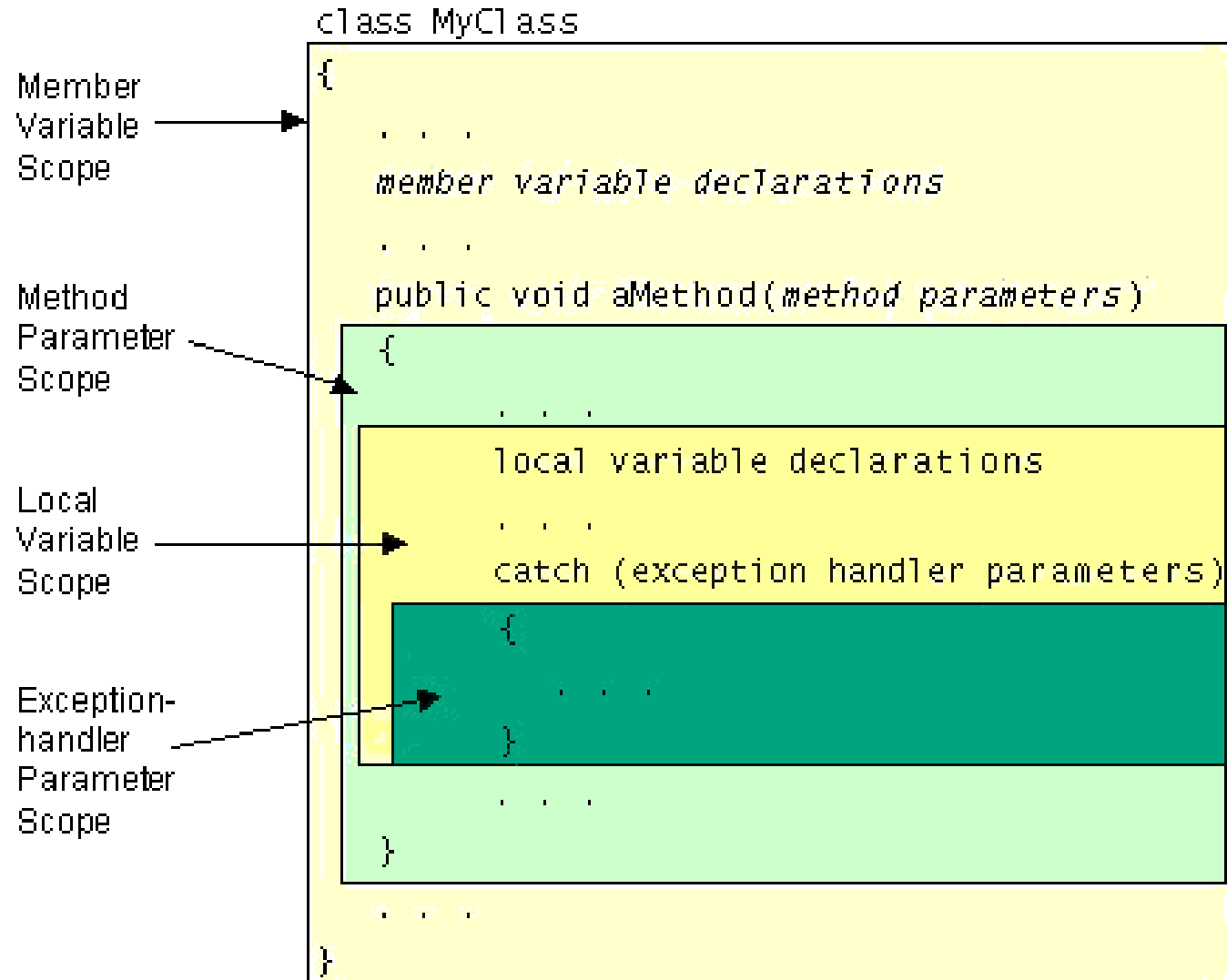


Explicit conversion
from **Employee**
reference to
SalesMan
reference (subclass
of Employee)

Scope Rules

- Scope
 - Portion of the program that can reference an identifier
 - Class scope
 - Begins at opening brace ({) and ends at closing brace (})
 - e.g., methods and instance variables
 - Block scope
 - Begins at identifier declaration and ends at closing brace (})
 - e.g., local variables and method parameters

Scope of variables



Method Overloading

- Method overloading
 - Several methods in the same class may share the same name

```
class DataRenderer {  
    void draw(String s) {  
        draw(5);  
        draw(5L);  
    }  
    void draw(int i) {  
        // . . .  
    }  
    void draw(long f) {  
        // . . .  
    }  
}
```


Method Overloading

- Overloaded methods are differentiated by
 - Number of parameters
 - Parameter types
- method *signature*
 - `draw(String s)` and `draw(String t)` have identical signature and result in a compiler error
 - `int draw(String s)` and `void draw(String s)` have identical signature and result in a compiler error

MethodOverload.java

```
// MethodOverload.java - Using overloaded methods
```

```
public class MethodOverload {  
  
    // main function  
    public static void main(String args[]) {  
        MethodOverload methodOverload = new MethodOverload();  
        methodOverload.square(7);  
        methodOverload.square(7.5);  
    }  
  
    // square method with int argument  
    public int square(int intValue) {  
        System.out.println(  
            "Called square with int argument: " + intValue);  
        return intValue * intValue;  
    } // end method square with int argument  
  
    // square method with double argument  
    public double square(double doubleValue) {  
        System.out.println(  
            "Called square with double argument: " +  
doubleValue);  
        return doubleValue * doubleValue;  
    } // end method square with double argument  
} // end class MethodOverload
```

MethodOverload.java

- Output:

```
•Called square with int argument: 7  
•Called square with double argument: 7.5
```

MethodOverload.java

```
// MethodOverload.java - Overloaded methods with identical  
// signatures and different return types.  
  
public class MethodOverload {  
  
    // first definition of method square with double argument  
    public int square(double x) {  
        return x * x;  
    }  
  
    // second definition of method square with double argument  
    // causes syntax error  
    public double square(double y) {  
        return y * y;  
    }  
} // end class MethodOverload
```

```
MethodOverload.java:18: square(double) is already defined in  
MethodOverload  
    public double square( double y )  
                        ^  
MethodOverload.java:13: possible loss of precision  
found   : double  
required: int  
    return x * x;  
           ^  
2 errors
```

Return Type Covariance

- Note, however, that a method in a subclass may override a method in a superclass and return a more specific type:
 - Given a class MammalFactory with method
`public Mammal breathe()`
 - and a class HumanFactory which extends Mammal,
 - then Human may have method
`public Human breathe()`

Riddle

- You extend a class where there is chuckNorris method
- `protected void chuckNorris(){...}`
- What is the valid way to override it?
 1. `private void chuckNorris() {...}`
 2. `protected void chuckNorris() {...}`
 3. `void chuckNorris() {...}`
 4. `public void chuckNorris() {...}`
 5. You can't override Chuck Norris method

Riddle

- You extend a class where there is chuckNorris method
- `protected void chuckNorris(){...}`
- What is the valid way to override it?
 1. `private void chuckNorris() {...}`
 2. **`protected void chuckNorris() {...}`**
 3. `void chuckNorris() {...}`
 4. **`public void chuckNorris() {...}`**
 5. You can't override Chuck Norris method

Java marasmus

- Primitive type are not an objects
- So they don't have methods
- You can't add them to Collection or Array of objects
- Wrapper classes exists to solve it

Wrapper Classes

- Type-wrapper classes
 - Each primitive type has one
 - Character, Byte, Integer, Boolean, etc.
 - Represents primitives as Objects
 - Polymorphic processing of data types (e.g.: Integer, Short, Long all extend Number)
 - Declared as final
 - Many methods are declared static
 - Part of java.lang package

Wrapper Classes

Some of the wrappers' functionality:

- Hold max and min values for type
- Convert wrappers to/from primitives
 - Since Java 5, this can be done implicitly using “autoboxing”
- Convert primitives/wrappers to/from Strings
- Can be used in object-based data structures

The Wrapper Classes: Example

- Convert an Integer to a String

```
Integer n = new Integer(10);  
String s = n.toString();
```

- Convert a String to an Integer

```
int i = Integer.parseInt("10"); // might throw an exception
```

Auto-boxing and -unboxing

Java 5 converts primitives to wrapper types as needed and vice versa

- Math intensive code should take extra care to avoid boxing

```
// boxing and unboxing of an int literal  
Integer wrapper = 0;  
int primitive = wrapper;
```

```
// boxing and unboxing of an int variable  
int primitive1 = 0;  
Integer wrapper = primitive1;  
int primitive2 = wrapper;
```

Task

- Write a class `GuessGame` with `play(int max)` and `printBestScore()` methods
- Computer pick a random number less than max
- Then asks for user to guess it
- Use `JOptionPane.showInputDialog(...)` for input
- Computer says: to big or to small till the picked number will be guessed
- Best result will be stored in instance variable of `GuessGame` class
- Score will be calculated by formula: $\text{max}/\text{number of tries}$

Chapter 5

Some Basic Java Classes

Index

- Class Object
- Memory management – The Garbage Collector
- String, StringBuilder, StringBuffer
- StringTokenizer
- Math class

Class Object

- The `Object` class sits at the top of the class hierarchy tree in the Java platform.
- The `Object` class defines the basic state and behavior that all objects share, such as:

Class Object

- `boolean equals (Object)`
 - to compare oneself to another object,
 - Default implementation - returns true if the target and the argument refer to the same instance (`==`)
 - Reflexive: `x.equals(x) == true`
 - Symmetric: `x.equals(y) == y.equals(x)`
 - Transitive: `x.equals(y) && y.equals(z) → x.equals(z)`
 - For any non-null x, `x.equals(null) == false`
 - Must be consistent and stable for similar values

Class Object

- `int hashCode()`
 - Returns a value (derived from object state) to be used for hashing
 - Must be consistent and stable for similar values
 - Equal objects must always produce the same hash code
 - Should not change (otherwise bugs will happen)
 - When overriding `equals()`, make sure you override `hashCode()` as well
 - To maintain above constraints – they are relied upon by many libraries
 - Generate hashcode and equals with intellij: alt insert

Class Object

- `String toString()`
 - convert to string (string representation of the object)
 - Default implementation returns: `class@ hash code`
- `Class getClass()`
 - returns the class of the object
 - returns an object of type 'Class' which represents the object's class in run-time.
 - This method can't be overridden because it is declared as final.

The Object Class: Example

```
public class NamedObject extends Object {
    protected final String name;
    public NamedObject(String name) { this.name = name; }
    public String toString() {
        return getClass().getName() + "#" + name;
    }
    public boolean equals(Object object) {
        if (!(object instanceof NamedObject)) return false;
        NamedObject namedObject = (NamedObject) object;
        return name.equals(namedObject.name);
    }
}

class Person extends NamedObject {
    Person(String name) { super(name); }
    // ...
}

Person p1 = new Person("John Smith");
Person p2 = new Person("John Smith");

System.out.println(p1); // Person#John Smith
System.out.println(p1.equals(p2)); // true
System.out.println(p1.equals(chickenSalad)); // false
```

The Object Finalization

- The Object class provides a method, (Like destructor)

```
void finalize()
```

- It's purpose is - cleaning up resources used by an object before it is garbage collected
 - A class that needs to free resources overrides this method (in theory)
 - Example - closing files that have been opened by the object.

The Object Finalization

- The `finalize` method is called automatically by the system
 - when an object is about to be collected by the garbage-collector
- Subclass `finalize` method
 - should invoke superclass `finalize` method
- Never use it (I'll tell you later why)

Memory Management

- Garbage Collector:
 - Periodically frees memory used by objects that are no longer needed
 - No 'delete' or 'free' operations are necessary
 - Scans dynamic memory areas for objects and marks those that are referenced
 - The garbage collector runs in a low-priority thread
 - Except for rare 'stop-the-world' pauses



Memory Management

- garbage collection can be *suggested* by `System.gc()`, but is hardly ever useful
 - Just let the garbage collector do its own thing!

String, StringBuilder

Three classes for storing and manipulating character data:

- `String` for immutable strings,
- `StringBuilder` for “mutable strings”
 - though not directly related to the `String` class
- `StringBuffer` is like `StringBuilder`
 - Guaranteed to be thread-safe, but slower
 - Might see it in older code, where it's okay
 - Prefer `StringBuilder` (there are better ways for obtaining thread-safety when needed)

Class String

Many constructors, here are a few:

```
String s = "hello"; // literal: not c-tor but returns String

// String s = ...
new String(); // new empty string
new String("hello"); // new copy of "hello"
new String(someStringRef); // copies string

// String isn't a character array,
// but can be built from one:
char charArray[] = { 'b', 'i', 'r', 't', 'h', ' ',
                     'd', 'a', 'y' };

// String s = ...
new String(charArray); // from char-array
new String(charArray, 6, 3); // char-array subset, "day"
```

Class String

- Method `length()`
 - Determine `String` length
 - Like arrays, `Strings` always “know” their size
 - Length is in characters, `!=` length in bytes
- Method `charAt`
 - Get character at specific location in `String`
- Method `getChars`
 - Get entire set of characters in `String`

Class String

Comparing `String` objects

- Method `equals`
 - Method `equals` tests two objects for equality using *lexicographical comparison*
- `'=='` operator
 - tests if both references refer to same object in memory

Class String

- Method `equalsIgnoreCase`
 - Test two objects for equality, but ignore case of letters in `String`
- Method `compareTo`
 - Compares `string` objects
 - Like `strcmp(C)`: $p1 < p2 \rightarrow \text{result} < 0$, etc.
- Method `toUpperCase/toLowerCase`
 - Возвращает тот же стринг, только маленькими или большими буквами соответственно

Comparing Strings

```
// 'asserts' fails if the argument is false  
assert "hello".equals("hello"); // succeeds  
assert "hello".equals("Hello"); // fails  
assert "hello".equalsIgnoreCase("Hello"); // succeeds  
assert "hello".regionMatches( 0, "Hello", 0, 5 ); // fails  
assert "hello".regionMatches( 1, "Hello", 1, 4 ); // succeeds  
assert "abc".compareTo("def") < 0; // succeeds  
assert "test".compareToIgnoreCase("Test") == 0; // succeeds
```

Concatenation and the + Operator

- Ways of concatenating Strings:

```
// String.concat()  
String helloString = "Hello Java";  
System.out.println(helloString.concat("World"));  
  
// + operator concat()  
String javaString = "Java";  
System.out.println("Hello" + javaString + "World");
```

Concatenation and the + Operator

- In both ways the concatenated strings do not change
 - A new string object is created in memory with the result of concatenation.

Concatenation and the + Operator

- It is possible that an operand of + is not a String:

```
System.out.println("My favorite number is " + 73);
```

- Autoboxing and toString() are called implicitly

Class `StringBuilder`

- Class `StringBuilder`
 - When `String` object is created, its contents cannot change
 - Used for creating and manipulating dynamic string data
 - i.e., modifiable `Strings`
 - Can store characters based on capacity
 - Capacity expands dynamically to handle additional characters
 - `String` operators `+` and `+=` actually rely on `StringBuilder` for concatenation

StringBuilder: Manipulation

```
StringBuilder builder = new StringBuilder("0123456789");
System.out.println(builder + " * " + builder.length());
builder.append("0123456789");
System.out.println(builder + " * " + builder.length());
builder.setLength(5);
System.out.println(builder + " * " + builder.length());
builder.append("999");
System.out.println(builder + " * " + builder.length());
```

0123456789 * 10

01234567890123456789 * 20

01234 * 5

01234999 * 8

StringBuilder

- Because append method returns reference on the same object you can write:
- `sBuilder.append("Pink Floyd, ").append("Therion")...`

Class StringTokenizer

- The string tokenizer class allows an application to break a string into tokens.

```
StringTokenizer tokenizer =  
    new StringTokenizer("this is a test");  
  
while (tokenizer.hasMoreTokens()) {  
    System.out.println(tokenizer.nextToken());  
}
```

- prints the following output:

```
this  
is  
a  
test
```

Class Math

- Class `java.lang.Math`
 - Contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions
 - All methods are static
 - no need of creating an object
 - using methods: `Math.method()`

Class Math

- Examples:
 - Calculate the square root of 900.0:
 - `Math.sqrt(900.0)`
 - Java random-number generators
 - `Math.random()`
 - `(int) (Math.random() * 6)`
 - Produces integers from 0 - 5

Chapter 6

Inheritance and Polymorphism

Index

- Inheritance
- Class casting
- Final classes
- Super reference
- Abstract classes and methods
- Interfaces
- Nested and inner classes
- Enums

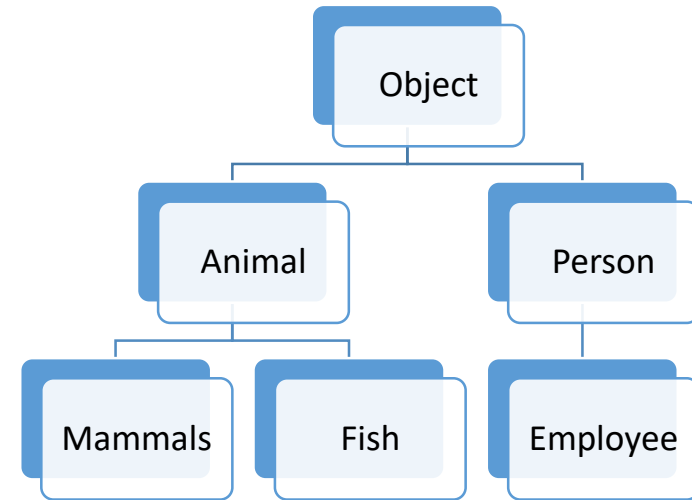
Inheritance

- In Java -
 - inheritance is declared in the class definition:

```
class Subclass extends DirectSuperClass {  
    // class body  
}
```

Inheritance

- Java's class hierarchy has one root - Class Object
 - All classes are subclasses (direct or indirect) of class Object
 - All classes in Java except for class Object have a superclass.
 - Every class has one and only one immediate superclass
 - No multiple inheritance!



Inheritance

- All classes in Java share basic behavior
 - Class Object Implements behavior that every class in the Java system needs
- If the class declaration has no extends clause, then the class has the class Object as its implicit direct superclass.

```
// Inheriting from Object implicitly  
class MyClass {  
    // class body  
}
```

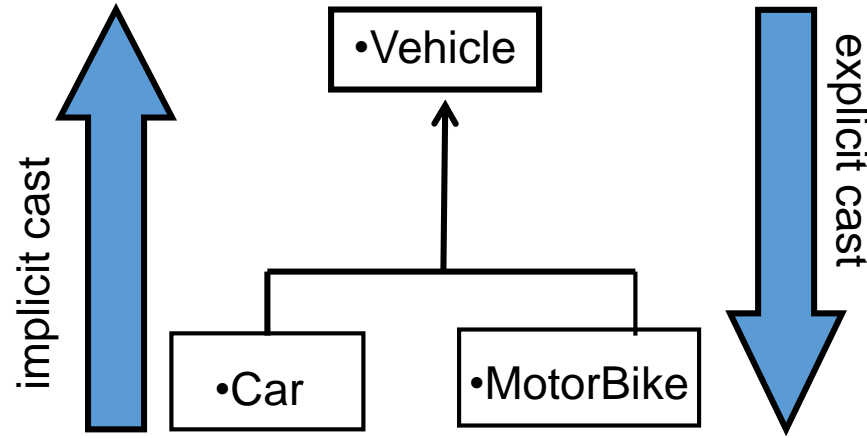
Inheritance: Example

```
class Vehicle {
    boolean needsFuel = true;
    void fillUp () { /*...*/ }
    void stop() {
        System.out.println("stopping");
        if (needsFuel) fillUp();
    }
}

class MotorBike extends Vehicle {
    String model;
    void fillUp() {
        System.out.println("Refuelling MotorBike");
    }
}

class Car extends Vehicle {
    void fillUp() {
        System.out.println("Refuelling Car");
    }
    void useSeatBelt() { /*...*/ }
}
```

Class Casting



```
MotorBike myBike = new MotorBike();
Vehicle vehicle = myBike;           // up cast
vehicle.fillUp();                   // MotorBike.fillUp
/* vehicle.useSeatBelt(); */        /* compilation error,
                                     no such method in Vehicle */

/* myBike = vehicle; */             /* compilation error
                                     explicit cast needed */

myBike = (MotorBike) vehicle;       // down cast
myBike.fillUp();                   // same MotorBike.fillUp as before
Car car = new Car();
vehicle = car;                     // up cast
myBike = (MotorBike) vehicle;       // java.lang.ClassCastException
```

Inheritance

- Inheritance can be prevented by declaring a class as `final`
- Method overriding can be prevented by declaring a method as `final`
- Method overriding can also be prevented by setting method visibility to `private`
 - Overriding in general is affected by visibility: you can only override what you can see...

Inheritance

- Final Classes
 - Classes which are impossible to inherit
 - Some of the basic classes of the language defined as final:
 - System, String, Boolean, Character, Byte, Short, Integer, Float, Double, etc

Why final?

- Purpose
 - Security Reasons
 - In classes that implement critical functionality, there is a danger that malicious subclass instances would pose as the base class instances and threaten system security.
 - Design reasons
 - Some programming interfaces aren't designed for subclassing at all and actively prevent it.
 - In many programming interfaces, it makes sense to extend some methods but not to extend other.

Super

- The `super` keyword - allows the sub-class to access members of the super-class
 - Used when a member is overridden
 - The method can refer to the hidden variables by using the `super` keyword.
 - The method can invoke the overridden method by using the `super` keyword.

Referring to Overridden Members using super

```
class ASillyClass {  
    boolean aVariable;  
    void aMethod() {  
        aVariable = true;  
    }  
}  
  
class ASillierClass extends ASillyClass {  
    boolean aVariable;  
    void aMethod() {  
        aVariable = false;  
        super.aMethod();  
        System.out.println(aVariable);  
        System.out.println(super.aVariable);  
    }  
}
```

ASillierClass's aMethod() prints:

false
true

Super

- The `super` keyword is used in constructors to call the super-class constructor
 - Constructors are not inherited
 - `super (. . .)` must be the first instruction
 - Default super constructor is called implicitly if not explicit call.

Super

```
class Vehicle {
    int initialSpeed;
    public Vehicle(int initialSpeed){
        this.initialSpeed = initialSpeed;
    }
    public Vehicle(){
        initialSpeed = 0;
    }
}

class Car extends Vehicle {
    boolean isAutomatic;
    public Car(boolean isAutomatic, int initialSpeed){
        super(initialSpeed); // calls Vehical's constructor
        this.isAutomatic = isAutomatic;
    }
    public Car(){ // super() is called implicitly
        isAutomatic = false;
    }
}
```

Abstract Classes and Methods

- An abstract class represents an abstract concept and so cannot be instantiated
- Use the keyword `abstract` before the `class` keyword in your class declaration to declare an abstract class :

```
abstract class Number {  
    . . .  
}
```

- If a class contains one abstract method, it must be an abstract class

Abstract Methods

- An abstract class may contain abstract methods as well as concrete methods.
- An abstract method has a signature, the keyword `abstract` and no implementation

```
abstract void draw();
```

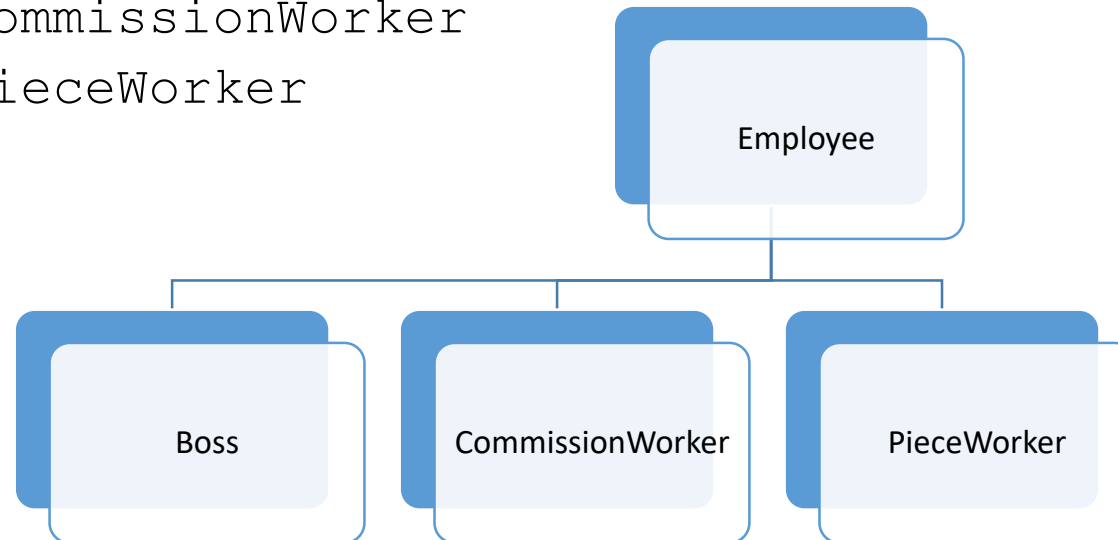
- Subclass must provide implementation
 - Otherwise it is also an abstract class

Example: Abstract Class and Methods

```
abstract class GraphicObject {  
    int x, y;  
    // . . .  
    void moveTo(int newX, int newY) {  
        // . . .  
        draw();  
    }  
    abstract void draw();  
}  
  
class Circle extends GraphicObject {  
    void draw() { /*...*/ } //Concrete implementation  
}  
  
class Rectangle extends GraphicObject {  
    void draw() { /*...*/ } //Concrete implementation  
}
```


Example: A Payroll System

- Abstract methods and polymorphism
 - Abstract superclass `Employee`
 - Method `earnings` applies to all employees
 - Person's earnings depends on type of `Employee`
 - Concrete `Employee` subclasses declared `final`
 - `Boss`
 - `CommissionWorker`
 - `PieceWorker`



Example: Employee.java

// Employee.java - Abstract base class Employee.

```
public abstract class Employee {  
    private String firstName;  
    private String lastName;  
  
    // constructor  
    public Employee(String first, String last) {  
        firstName = first;  
        lastName = last;  
    }  
  
    public String getFirstName() { return firstName; }  
  
    public String getLastName() { return lastName; }  
  
    public String toString() {  
        return firstName + " " + lastName;  
    }  
  
    public abstract double earnings();  
} // end class Employee
```

Example: Bos.java

// Boss.java - Boss class derived from Employee.

```
public final class Boss extends Employee {
    private double weeklySalary;

    // constructor for class Boss
    public Boss(String first, String last, double salary) {
        super(first, last); // call superclass constructor
        setWeeklySalary(salary);
    }

    // set Boss's salary
    public void setWeeklySalary(double salary) {
        weeklySalary = salary;
    }

    // get Boss's pay
    public double earnings() { return weeklySalary; }

    // get String representation of Boss
    public String toString() {
        return "Boss: " + super.toString();
    }
} // end class Boss
```

Example: CommissionWorker.java

// CommissionWorker.java

```
public final class CommissionWorker extends Employee {
    private double salary;           // base salary per week
    private double commission;       // amount per item sold
    private int quantity;            // total items sold for week

    // constructor for class CommissionWorker
    public CommissionWorker(String first, String last,
                           double salary, double commission,
                           int quantity) {
        super(first, last); // call superclass constructor
        setSalary(salary);
        setCommission(commission);
        setQuantity(quantity);
    }

    // set CommissionWorker's weekly base salary
    public void setSalary(double weeklySalary) {
        salary = weeklySalary;
    }
}
```

Example: CommissionWorker.java

```
// set CommissionWorker's commission
public void setCommission(double itemCommission) {
    commission = itemCommission;
}

// set CommissionWorker's quantity sold
public void setQuantity(int totalSold) {
    quantity = totalSold;
}

// determine CommissionWorker's earnings
public double earnings() {
    return salary + commission * quantity;
}

// get String representation of CommissionWorker's name
public String toString() {
    return "Commission worker: " + super.toString();
}
} // end class CommissionWorker
```

Example: PieceWorker.java

// PieceWorker.java - PieceWorker class derived from Employee

```
public final class PieceWorker extends Employee {  
    private double wagePerPiece; // wage per piece output  
    private int quantity; // output for week  
  
    // constructor for class PieceWorker  
    public PieceWorker(String first, String last,  
                       double wage, int numberOfItems) {  
        super(first, last); // call superclass constructor  
        setWage(wage);  
        setQuantity(numberOfItems);  
    }  
  
    // set PieceWorker's wage  
    public void setWage(double wage) {  
        wagePerPiece = wage;  
    }  
}
```

Example: PieceWorker.java

```
// set number of items output
public void setQuantity(int numberOfItems) {
    quantity = numberOfItems;
}

// determine PieceWorker's earnings
public double earnings() {
    return quantity * wagePerPiece;
}

public String toString() {
    return "Piece worker: " + super.toString();
}
} // end class PieceWorker
```

Interfaces

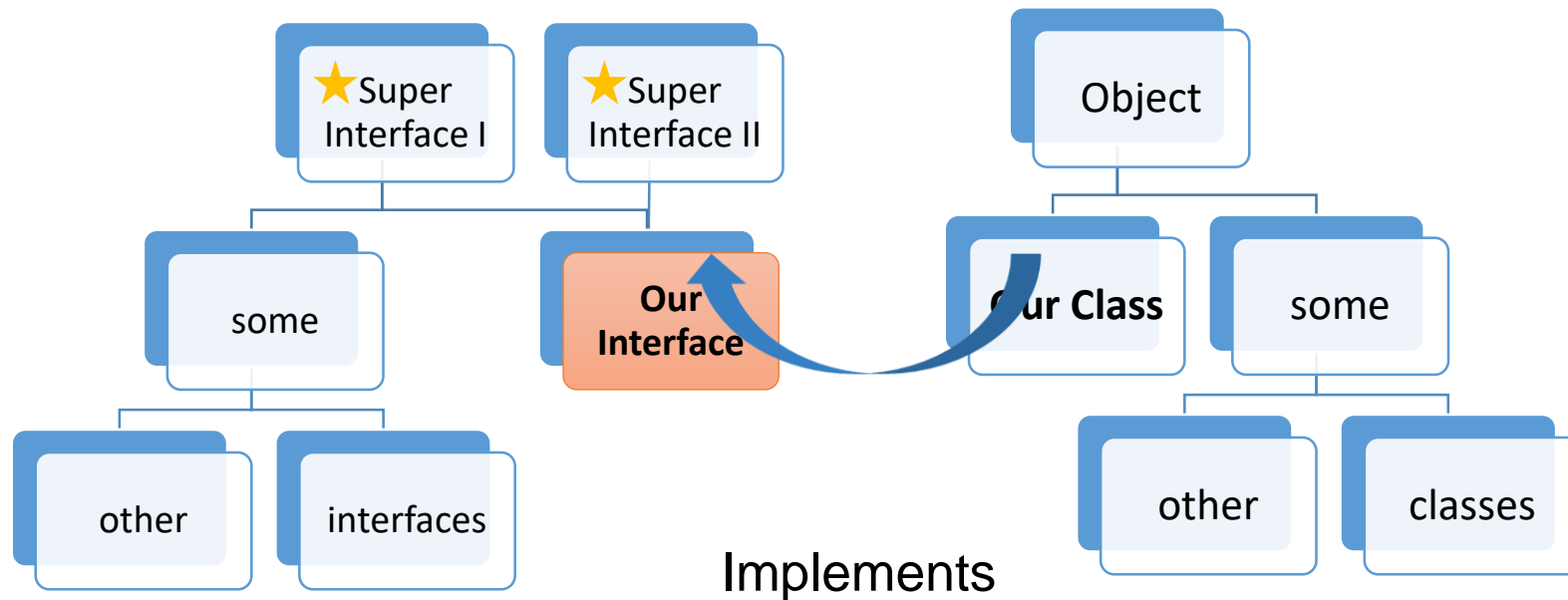
- An interface is a set of method definitions without implementations
 - All methods in an interface are (implicitly)
`abstract public`
- An interface can also include declarations of constants
 - All constants are (implicitly)
`public static final`

Interface

- An Interface may be `public` or default (package)
- A class can implement any number of interfaces - in addition to extending one class.
 - A concrete class must implement all interface methods
 - Otherwise it must be abstract

Interface

- An interface may extend other interfaces
 - 0 to many



•Hierarchy of Interfaces

•Hierarchy of Classes

Interface

- An interface declaration introduces a new *reference type*

```
InterfaceType interfaceType = new  
    ClassWhichImplementsThisInterface()
```

- You may define a collection of objects that implement a certain interface.
 - Example:

```
ActionEvent eventsList[]
```

Example: Interface

```
interface Printable {  
    void print();  
}  
  
class Point extends Object implements Printable {  
    private double x, y;  
    //...  
    // must be public:  
    public void print() {  
        System.out.println(x);  
        System.out.println(y);  
    }  
}
```

Interface: Multiple Inheritance

- An interface may extend many interfaces
- A class can implement many interfaces

```
interface Printable {  
    void print();  
}  
interface Stream {  
    int read();  
}  
interface DataStream extends Stream {  
    float readFloat();  
}  
class MyStream implements DataStream, Printable {  
    public int read() { /*...*/ return 0; }  
    public float readFloat() { /*...*/ return 0f; }  
    public void print() { /*...*/ }  
}
```

Interfaces and Types

- Interfaces are similar to other kinds of types
 - Interface names can be used wherever other (abstract) type names are used.
 - The `instanceof` operator may be used to check if an object implements a given interface

```
Point point = new Point();  
if (point instanceof Printable) {  
    Printable p = (Printable)point;  
}
```

Interfaces and Variables

- Interface variables are `public`, `static` and `final`
 - Constants
- Like all constants, must be initialized

```
interface MyInterface {  
    // all these effectively have the same modifiers:  
    public static final int ONE = 1;  
        static final int TWO = 2;  
            final int THREE = 3;  
        static      int FOUR = 4;  
            int FIVE = 5;  
  
    // etc ...  
}  
  
int i = MyInterface.ONE;
```

Since Java 5 enums are used instead

Lab – Write heroes game in text mode

Class Character int power, int hp void kick(Character c) boolean isAlive()

Hobbit power =0, hp = 3, kick(toCry());)

Elf hp = 10, power = 10, kick(kill everybody which weaker than him, otherwise decrease power of other character by 1)

King power 5-15, hp 5-15, kick(decrease number of hp of the enemy by random number which will be in range of his power

Knight power 2-12, hp 2-12, kick(like King)

CharacterFactory

Character createCharacter() – returns random instance of any existing character

GameManager

void fight(Character c1, Character c2){

to provide fight between to characters and explain via command line what happens during the fight, till both of the characters are alive

}

Nested and Inner Classes

- A nested class is a class that is a member of another class.
 - A nested class defined within it's enclosing class.
 - Example:

```
public class Tree {  
    // ...  
    class Leave {  
        // ...  
    }  
}
```

Nested and Inner Classes

- Purpose
 - Used to express and enforce a containment relationship between two classes
 - one contains the other.
 - when a class only makes sense in the context of its enclosing class or when it relies on the enclosing class for its function.

Nested and Inner Classes

- A static nested class is called just that: a static nested class
 - lexically scoped by the outer class, but with no access to instance data
- A non-static nested class is called an “inner class”

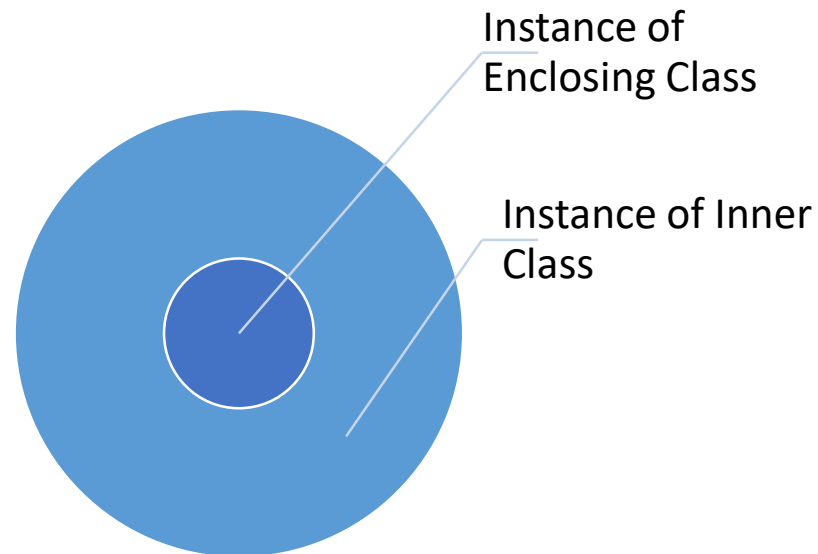
```
class EnclosingClass {  
    // ...  
  
    static class AStaticNestedClass { /* ... */ }  
  
    class InnerClass { /* ... */ }  
}
```

Nested and Inner Classes

- Static nested class characteristics:
 - Also have direct access to the outer class's members – but only the static ones.

Nested and Inner Classes

- Definition: An *inner class* is a nested class
 - whose instance exists within an instance of its enclosing class, and
 - has direct access to the instance members of its enclosing instance.



Nested and Inner Classes

- Inner class Characteristics:
 - The inner class has direct access to all the outer class's members
 - including private members
 - and to the outer class's 'this' and 'super' – OuterClassName.this
 - because an inner class is associated with an instance, it cannot define any static members itself

Nested and Inner Classes

- The outer class is responsible for creating objects of its inner classes
- To create an object of an inner class outside the outer class:

```
OuterClassName outer =  
    new OuterClassName();  
  
OuterClassName.InnerClassName y =  
    outer.new InnerClassName();
```

Anonymous inner class

- Inner class without name
- Created where a class is defined at usage site

```
ActionListener listener = new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        // ...  
    }  
}
```

- This code defines an anonymous inner class, which implements an interface called ActionListener.
- The code also creates an instance of this unnamed class.

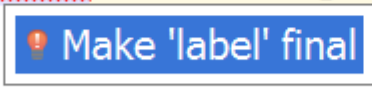
Riddle

```
public class MyFrame extends JFrame {  
    public MyFrame() {  
        Label label = new Label();  
        JButton button = new JButton("click to change color");  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                label.setBackground(Color.RED);  
            }  
        });  
    }  
}
```

Will it compile?

Just add final (Intelij will do it automatically)

```
public class MyFrame extends JFrame {  
    public MyFrame() {  
        Label label = new Label();  
        JButton button = new JButton("click to change color");  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                label.setBackground(Color.RED);  
            }  
        });  
    }  
}
```

A blue tooltip with a red exclamation mark icon and the text "Make 'label' final" is positioned over the line containing the `label.setBackground` call.

Since Java 8 it is redundant. Java 8 is «so cool» that it can add final where it necessary in compile time 😊

Java 8 is the the best

She knows to add final without request

Nested and Inner Classes

- Any nested class, can be declared in any block of code.
 - A nested class declared within a method or other smaller block of code has access to any final, local variables in scope.

Nested and Inner Classes

- Compiling a class that contains inner class
 - Results in separate `.class` file
 - inner classes with names:
 - `OuterClassName$InnerClassName.class`
 - anonymous inner classes
 - `OuterClassName$n.class` ($n=1, 2, \dots$)

Lab

Make JFrame like one from the picture:
With JButton and JTextField.

When button will be clicked text from
Text field should be displayed as a title

You need to use:

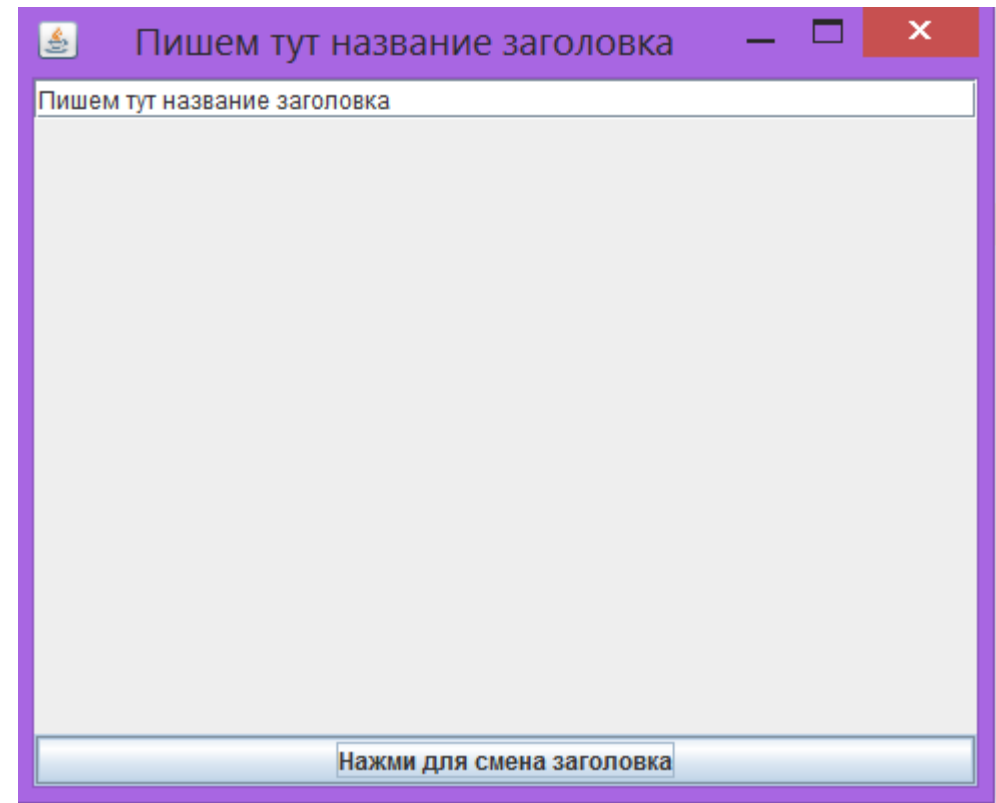
JFrame, JTextField, JButton

In order to add component on frame you should apply to its contentPane

By default it has BorderLayout

Example:

```
getContentPane().add(button, BorderLayout.SOUTH);
```



I have a question

- You write Person class.
- You need to add attribute marital status (single, married...)
- What type will it be? String? Int? Some custom class MaritalStatus?

Enums

- Used to define a finite set of allowed values for a specific data type
- Season may be only WINTER, SPRING, SUMMER or AUTUMN
- Class-like structure, named enum
- Must include list of allowed values
- Optionally may include:
 - An interface or set of interfaces that the enum implements
 - Variable definitions
 - Method definitions
 - Value-specific class bodies

Basic Enum Example

- Declaration:

```
enum Season{ WINTER, SPRING, SUMMER, AUTUMN};
```

- Usage:

```
class Month {  
    private final String name;  
    private final Season season;  
    public Month(String name, Season season){  
        this.name = name;  
        this.season = season;  
    }  
    // ...  
}  
  
// ...  
Month jan = new Month("January", Season.WINTER);
```

Enums explained

- Enums are classes
 - Type-safety
 - Compile-time checking
- Enums implicitly extend `java.lang.Enum`
 - Which overrides some `Object` behavior E.g. `toString()`
 - And adds couple useful methods, which doesn't make sense for regular class. E.g. `values()` – which returns array of all objects with type of the enum

Enums explained

- Enumerated types aren't integers, but an instance of the specialized Enum class itself
- Enums have no *public* constructors to prevent the ability to create additional instances
- Enum values are public, static, and final
- The enum itself is effectively final, as it cannot be subclassed

Enums explained

- Enum values can be compared with `==` or `equals()`
- Enums implements `java.lang.Comparable`
- Enums override `toString()`
- Enums provide a `valueOf()` method, which complements `toString()`

Enums explained

- Enums define a final instance method named `ordinal()`
 - Returns the integer position of each enumerated value
 - Not for direct use in code – usually, that’s what you have the object for!
- Enums define a `values()` method, which allows for iteration over the values of an enum

Iteration on Enums

- Iteration by using values()
- It is legal to use switch

```
for (Season s : Season.values()) {  
    System.out.println("Season: " + s);  
}
```

```
switch (month.getSeason()) {  
    case WINTER:  
        out.println("cold!");  
        break;  
    case SUMMER:  
        out.println("hot!");  
        break;  
    case SPRING:  
    case AUTUMN:  
        out.println("nice!");  
        break;  
}
```

- No default, must consider all values.
- Leave the case values unqualified.

Enum value-specific class bodies

- It is possible to write different class body for every value:

```
enum Season {  
    WINTER() {  
        String[] getSports() {  
            return new String[]{"hockey", "skates", "ski"};  
        }  
    }  
    /* for every value ... */ ;  
  
    abstract String[] getSports();  
}
```

Lab

- Write enum MaritalStatus (MARRIED, SINGLE, DIVORCED, WIDOW)
- Create property: denrew Description with appropriate values
- Override toString(). When enum will be printed Hebrew description should be seen
- Create Person class with age and marital status
- Write class PersonReader which will read person data from client and will print it to cmd
- Implement toString in person like this: Evgeny Borisov age: 38, גשוי.
- When reading person data marital status will be number from 1 to 4
- In case such person is like previous one print: “You again??” (implement equals method)

Advanced Lab – write Enum for HTTP codes

- Informational (100 – 199)
- Success (200 – 299)
- 3xx Redirection (300 – 399)
- 4xx Client Error (400 – 499)
- 5xx Server Error (500 – 599)

Task – write Enum for HTTP code

Write Service which receives http code and it appropriately

Enum solution

```
public enum HttpStatusEnum {  
    INFORMATIONAL(100, 199, "תידע", new HttpInformationalHandler()),  
    SUCCESS(200, 299, "הצלחה", new HttpRegularHandler()),  
    REDIRECTION(300, 399, "ניתוב חודש", new HttpRedirectionalHandler()),  
    CLIENT_ERROR(400, 499, "שגיאה אצל לקוח", new HttpClientErrorHandler()),  
    SERVER_ERROR(500, 599, "שגיאה בשרת", new HttpServerErrorHandler());  
    private int max,min;  
    private String hebrewDescription;  
    private HttpHandler handler;  
    HttpStatusEnum(int min, int max, String hebrewDescription, HttpHandler handler) {  
        this.min = min;  
        this.max = max;  
        this.hebrewDescription = hebrewDescription;  
        this.handler = handler;  
    }  
    public static HttpStatusEnum getHttpStatus(int errorCode) {  
        for (HttpStatusEnum value : values()) {  
            if (errorCode >= value.min && errorCode < value.max) {  
                return value;  
            }  
        }  
        throw new ErrorCodeNotFoundException("unknown error code " + errorCode);  
    }  
    public String getHebrewDescription() {...}  
    public HttpHandler getHandler() {...}  
}
```

Enum test

```
public class HttpStatusCodeTest {  
    public static void main(String[] args) {  
        System.out.println(HttpStatusEnum.getHttpStatus(101).getHebrewDescription());  
        System.out.println(HttpStatusEnum.getHttpStatus(231).getHebrewDescription());  
        System.out.println(HttpStatusEnum.getHttpStatus(353).getHebrewDescription());  
        System.out.println(HttpStatusEnum.getHttpStatus(444).getHebrewDescription());  
        HttpStatusEnum.getHttpStatus(450).getHandler().handle();  
        System.out.println(HttpStatusEnum.getHttpStatus(502).getHebrewDescription());  
        System.out.println(HttpStatusEnum.getHttpStatus(651).getHebrewDescription());  
        System.out.println(HttpStatusEnum.CLIENT_ERROR);  
        System.out.println(HttpStatusEnum.SERVER_ERROR);  
        System.out.println(HttpStatusEnum.REDIRECTION);  
        System.out.println(HttpStatusEnum.SUCCESS);  
        System.out.println(HttpStatusEnum.INFORMATIONAL);  
    }  
}  
}
```

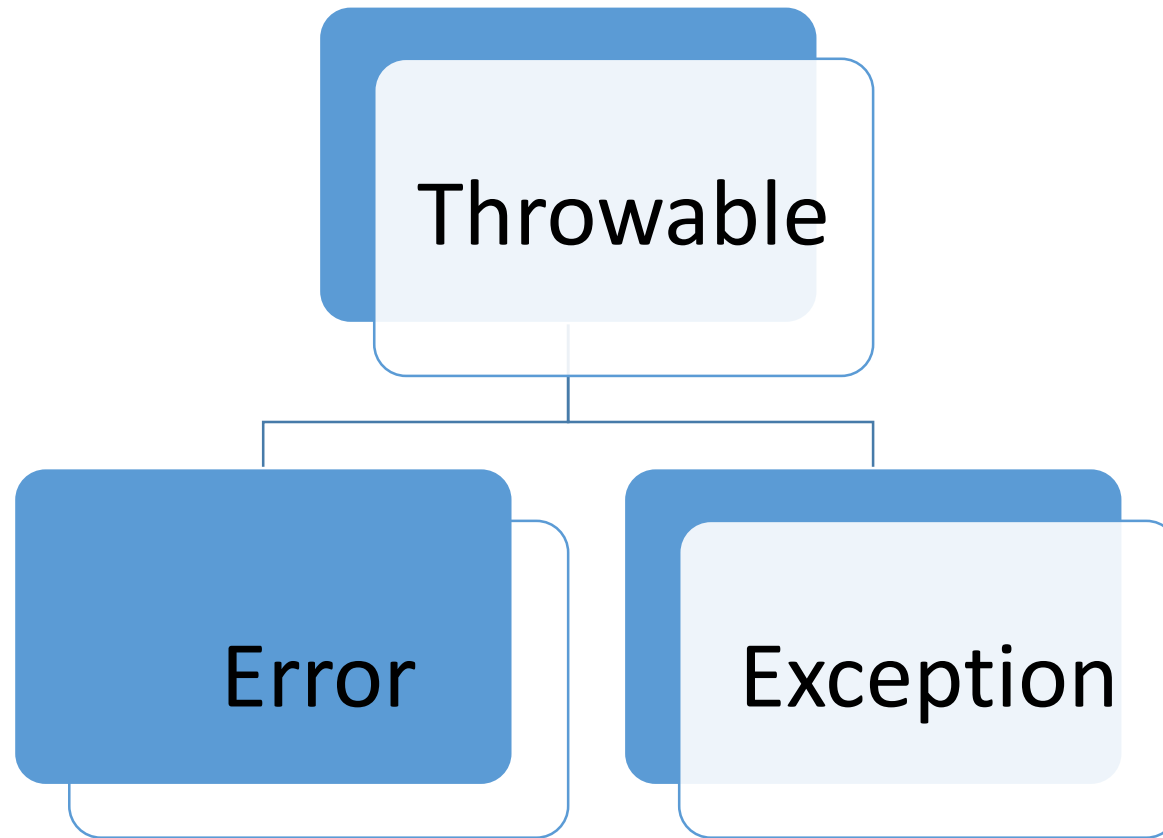
מידע
הצלחה
נietוב מחדש
טויה אצל לקוח
client error handle
טויה בשרת
Exception in thread "main" com.idi.learning.enums.ErrorCodeNotFoundException: unknown error code 651
 at com.idi.learning.enums.HttpStatusEnum.getHttpStatus([HttpStatusEnum.java:35](#))
 at com.idi.learning.enums.HttpStatusCodeTest.main([HttpStatusCodeTest.java:18](#)) <5 internal calls>

Exception Handling

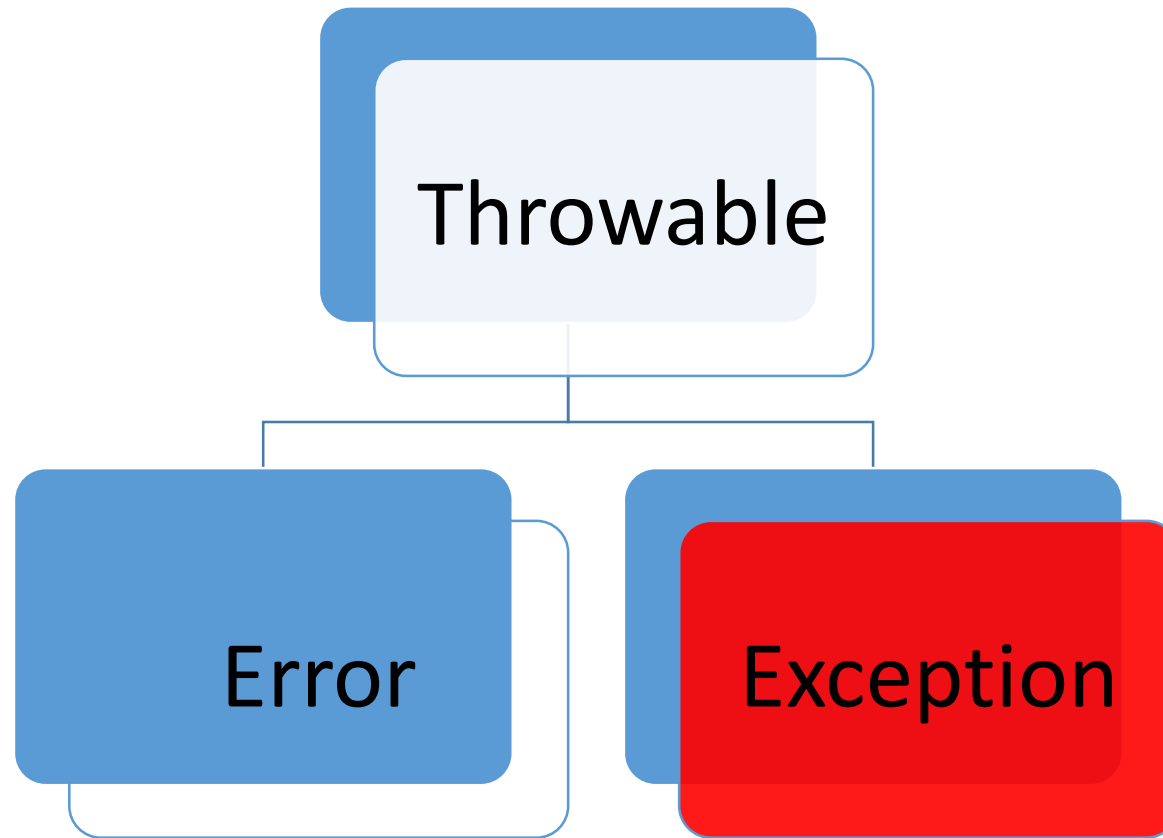
Introduction

- *Exceptions* are things that are not supposed to occur
- Some exceptions (like division by zero) are avoidable through careful programming
- Some exceptions (like losing a network connection) are not avoidable or predictable
- Java allows programmers to define their own means of handling exceptions when they occur

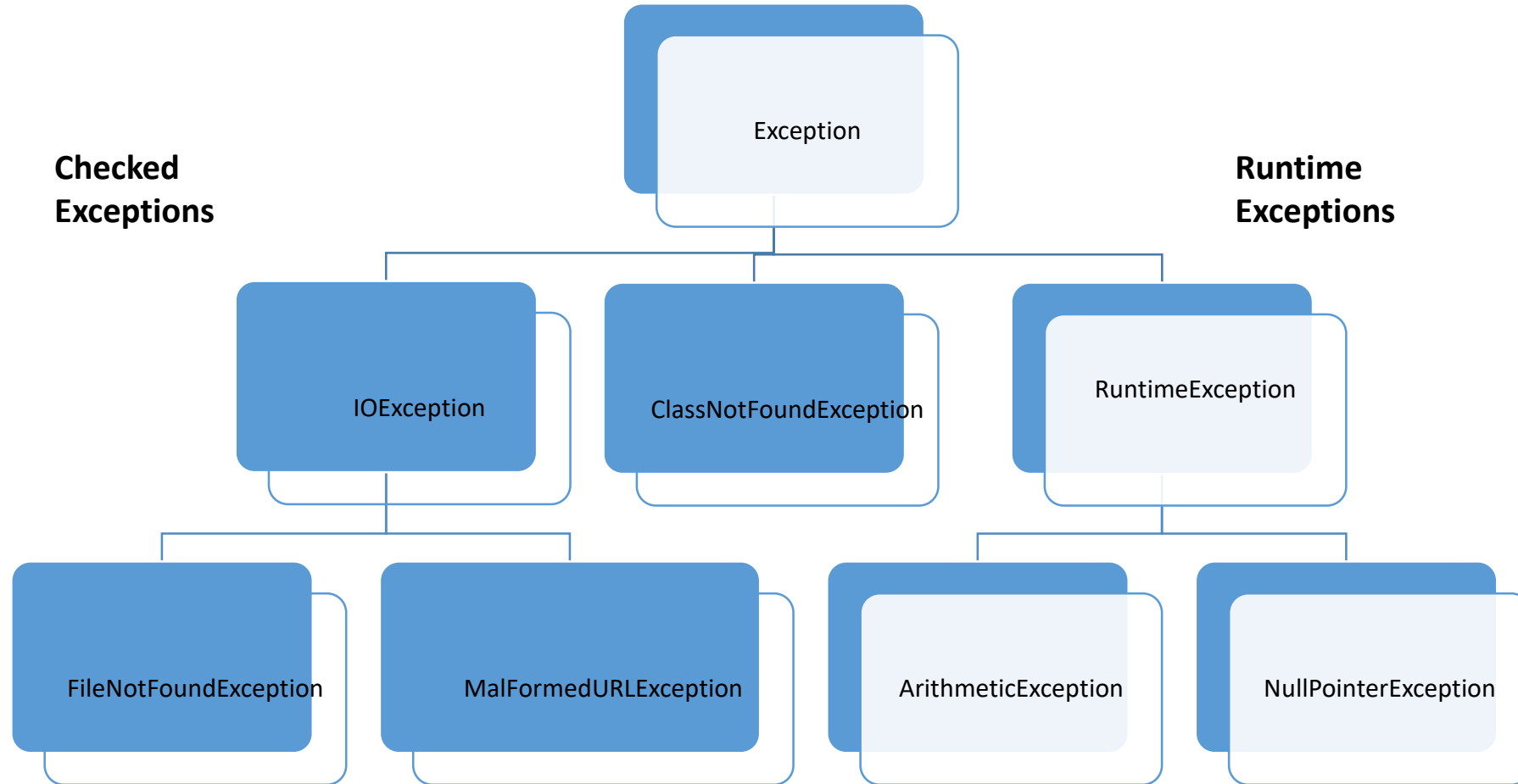
Type of exceptions



Type of exceptions



Type of exceptions



What is Exception Object

- Exception has several constructors:

[Exception](#)() [Exception](#)([String](#) message)

[Exception](#)([String](#) message, [Throwable](#) cause)

- Exception has several methods:

public [Throwable](#) **getCause**()

public [String](#) **getMessage**()

public void **printStackTrace**() /getStackTrace

- Exceptions state keeps info about exception – that is the main purpose of exception object.
- Exception name also reproduce information
- You can write your own exceptions classes

Method exception declaration

- Each method can be declared as method which throws an exception
- That means:
 1. Due to method execution an exception could be thrown
 2. One, who uses such a method MUST handle the possible exception
 3. If type of declared exception is RuntimeException it is **not necessary** to care about handling an exception

Keywords for Java Exceptions

- **throws**
Describes the exceptions which can be raised by a method.
- **throw**
Raises an exception to the first available handler in the call stack, unwinding the stack along the way.
- **try**
Marks the start of a block associated with a set of exception handlers.
- **catch**
If the block enclosed by the try generates an exception of this type, control moves here; watch out for implicit subsumption.
- **finally**
Always called when the try block concludes, and after any necessary catch handler is complete.

Handling an exception

1. Take care about the case, when exception happens in the same code block you run such a method
2. Declare your own method (in which you run a method who throws an exception) as method who throws the same, or more common (wide) exception.
This mean that one who will run your method should care about the exception handling

Declaring and throwing an exception

```
public boolean isTextExistsInFile(String text, File f) throws FileNotFoundException{  
    if(!f.exists()){  
        throw new FileNotFoundException("file not exists");  
    }  
    else{  
        //code  
    }  
}
```

This mean that such a method can possible throw an exception

One who calls this method, MUST handle the exception (because it is not a RuntimeException)

Handling an exception

```
try {  
    isTextExistsInFile("java",f);  
} catch (FileNotFoundException e) {  
    e.printStackTrace();  
}
```

try block must surround the part of the code which may throw an exception
in the catch block you write all your handling stuff.

In most of the cases it will be writing the exception details in to the log file or console

Handling an exception

```
try {  
    | isTextExistsInFile("java",f);  
} catch (FileNotFoundException e) {  
    | e.printStackTrace();  
}  
finally {
```

This code will run anyway, no matters whether exception occurred or not usually this block is used to close resources.
connection.close()... outputStream.close()... etc.

```
}
```


Handling multiple exception types

- If in the are several different type of exceptions may occur in the same code block the are 3 ways of handle:
 1. Break this block to several parts and surround each one with try and catch (ugly overloaded code)
 2. In case the handle will be the same no matter which kind of exception happens – surround all this code block with one try and catch an parent Exception.
 3. You can write more than one **catch** to one **try**. Starting from the narrow to more wide exception.
The handle works similar to **switch** statement which uses break after each **case**

Handling multiple exception types

```
File f = null;
try {
    isTextExistsInFile("java",f); //this line throws FileNotFoundException
    FileOutputStream fis = new FileOutputStream(f);
    ObjectOutputStream oos = new ObjectOutputStream(fis); //throws IOException
} catch (IOException e) {
    e.printStackTrace(); //same handle for both
}
```

Handling multiple exception types

```
File f = null;
try {
    isTextExistsInFile("java",f); //this line throws FileNotFoundException
    FileOutputStream fis = new FileOutputStream(f);
    ObjectOutputStream oos = new ObjectOutputStream(fis); //throws IOException
} catch (FileNotFoundException e) {
    // handle FileNotFoundException
} catch (IOException e){
    // handle IOException
}
```

What wrong here?

```
File f = null;
try {
    isTextExistsInFile("java",f); //this line throws FileNotFoundException
    FileOutputStream fis = new FileOutputStream(f);
    ObjectOutputStream oos = new ObjectOutputStream(fis); //throws IOException
} catch (IOException e) {
    // handle FileNotFoundException
} catch (FileNotFoundException e){
    // handle IOException
}
```

- Why it won't compile?
- Because FileNotFoundException extends from IOException (it is more specific type of IOException) so second catch is unreachable, the first one will always handle the exception.

It is ugly!

```
public String readLine(String path) {  
    BufferedReader br = null;  
    try {  
        br = new BufferedReader(new FileReader(path));  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    }  
    String line = null;  
    try {  
        line = br.readLine();  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            br.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
    return line;  
}
```

Since Java 7 you have try with resources

```
public String readLine(String path) throws IOException {  
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }  
}
```

- But it just solves finally problem when closing resources
- Classes which implements Closeable interface override close() method, which will be executed no matter exception

Riddle

- You want override method, which throws IOException
 - Which Exception can be added to signature of your method?
1. FileNotFoundException
 2. Exception
 3. NullPointerException
 4. Any

Riddle

- You want override method, which throws IOException
- Which Exception can be added to signature of your method?

1. **FileNotFoundException**

2. Exception

3. **NullPointerException**

4. Any

Conclusions

- You can narrow Exception scope, but not wide, so it is not always possible to add exception to method signature
- So sometimes you should add meaningless catches
- What should be written inside?

```
try {  
    FileReader fileReader = new FileReader("brodsky.txt");  
} catch (FileNotFoundException e) {  
    throw new RuntimeException(e);  
}
```

Wrapping the exception

- Always, **always, always!!!** keep the original exception
- This is done by passing its reference to the constructor of the new Exception
- By not doing so you cut the stack
 - Nobody will be able to track the exception source
- Do it. Keep the original. It is important.

Bad Exceptions Handling

- Empty catch clause
 - This is the worse. The **WORSE!**
 - Sometimes the reason may be “this can never happen”
 - In this case it even will be more important to know about the exception.
- Only `e.printStackTrace()`
 - We have already discussed it, the “Don’t put the head in sand” thing
 - In most cases it is **NOT GOOD ENOUGH**

Custom Exception

- Think if you really need it? Maybe you can use existing exception class
- Create a class which extends from some existing exception class (only exceptions can be thrown)
- Think if you want to extend checked or unchecked exception
- Extend RuntimeException
- Create appropriate constructor (alt insert in intellij)

Custom Exception example

```
public class WTFException extends RuntimeException {  
    public WTFException(String message) {  
        super(message);  
    }  
  
    public WTFException(String message, Throwable cause) {  
        super(message, cause);  
    }  
  
    public WTFException(Throwable cause) {  
        super(cause);  
    }  
}
```



Checked exceptions is a crime
God commandment:
Throw Runtime!

Chapter 7:

Date/Time Handling

Index

- Date
- Calendar
- Joda-Time

Standard Library Date Handling

- Two layers:
 - Date for representing... dates
 - Calendar for date related calculations

Date and DateFormat

- The `java.util.Date` class represents a specific instant in time, with millisecond precision
 - `public Date()`
 - Allocates a `Date` object and initializes it so that it represents the time at which it was allocated
- Most of the class is deprecated

Date and DateFormat

- The `java.text.DateFormat` class allows you to format dates and times with predefined styles in a locale-sensitive manner (4/20/98, 20.4.98, etc...)
- Abstract class
- provides static factory methods for obtaining date/time formatters

```
DateFormat dateFormat = DateFormat.getDateInstance();  
String dateString = dateFormat.format(date);
```

Date and DateFormat

- `class SimpleDateFormat`
 - inherits `DateFormat`
 - Concrete
 - The default format for current locale
 - returned by - `DateFormat.getDateInstance()` ;
 - most commonly used

Date and DateFormat

- To specify the time format use a *time pattern* string.
 - In this pattern, ASCII letters are reserved as pattern letters:

Symbol	Meaning	Example	
G	era designator	AD	
y	year		1996
M	month in year	July & 07	
d	day in month	10	
h	hour in am/pm (1~12)	12	
...			

Date and DateFormat

- Example:

```
Date date = new Date();
```

```
// Format the current time
```

```
SimpleDateFormat formatter =
```

```
    new SimpleDateFormat(
```

```
        "yyyy.MM.dd G 'at' hh:mm:ss a zzz");
```

```
Date t1 = new Date();
```

```
String dateString = formatter.format(t1);
```

```
// Parse the previous string back into a Date
```

```
// 'parse' returns null if the input could not be parsed
```

```
ParsePosition pos = new ParsePosition(0);
```

```
Date t2 = formatter.parse(dateString, pos);
```

Date and DateFormat

- By default - system fetches the information about the local language and country conventions

- To format for a different Locale, specify it in a `getDateInstance()` call

```
DateFormat dateFormat = DateFormat.getDateInstance ( DateFormat.LONG, Locale.FRANCE );
```

Date and DateFormat

- You can pass in different options to these factory methods to control the length of the result
 - SHORT
 - MEDIUM
 - LONG
 - FULL

Date and DateFormat

- Sample Date Formats

Style	U.S. Locale	French Locale
DEFAULT	10-Apr-98	10 avr 98
SHORT	4/10/98	10/04/98
MEDIUM	10-Apr-98	10 avr 98
LONG	April 10, 1998	10 avril 1998
FULL	Friday, April 10, 1998	vendredi, 10 avril 1998

Date and Calendar

- Get your dates from Calendar objects:
 - Calendar's `getTime()` method returns a Date
- Don't use Date's deprecated methods

Calendar

The Calendar API offers some strong date and time handling capabilities:

```
// get a calendar with current date/time  
// (for current locale and time zone)  
// then set the calendar to tomorrow's date  
Calendar tomorrow = Calendar.getInstance();  
tomorrow.add(Calendar.DATE, 1);  
  
// today...  
Calendar today = Calendar.getInstance();  
  
if (tomorrow.after(today)) {  
    System.out.println("yup...");  
}
```

Calendar

- Relies on “`today`”

```
today.set(Calendar.HOUR_OF_DAY, 23);  
assert today.get(Calendar.AM_PM) == Calendar.PM;
```
- This gets cumbersome after a while
 - DAY, DAY_OF_WEEK, HOUR, HOUR_OF_DAY, DAY_OF_MONTH, DAY_OF_YEAR,
...
- Offers good services on calendar fields
- Offers no services for other date/time related arithmetics.

Joda Time

Java date and time API

Joda Time

- Joda-Time provides a quality replacement for the Java *date* and *time* classes
- The design allows for multiple calendar systems, while still providing a simple API.
- Supporting the Gregorian, Julian, Buddhist, Coptic, Ethiopic and Islamic calendar.
- Supporting classes include time zone, duration, format and parsing.

Why Joda Time?

- Joda-Time has been created to radically change **date** and **time** handling in Java. The JDK classes Date and Calendar are very badly designed, have had numerous bugs and have odd performance effects

JDK Date

`java.util.Date`

- Wasn't updated (from JDK1.0)
- Uses two digit years (from 1900)
- January is 0, December is 11
- Not immutable.
- Not Threadsafe.
- Most methods deprecated in JDK1.1
- Uses milliseconds from 1970 representation

Advantages of Joda Time

- **Easy to Use:** simple field accessors like such as `getYear()` or `getDayOfWeek()`.
- **Easy to Extend.** Joda-Time supports multiple calendar systems via a pluggable system based on the Chronology class.
- **Comprehensive Feature Set.** The library is intended to provide all the functionality that is required for date-time calculations. It already provides out-of-the-box features, such as support for oddball date formats, which are difficult to replicate with the JDK.
- **Up-to-date Time Zone calculations.** The time zone implementation is based on the public tz database which is updated several times a year. Example: `DateTimeZone.forID("Europe/London")`
- **Calendar support.** The library currently provides 8 calendar systems. More will be added in the future

Advantages of Joda Time

- **Easy interoperability.** The library internally uses a millisecond instant which is identical to the JDK and similar to other common time representations.
- **Better Performance Characteristics.** Joda-Time does only the minimal calculation for the field that is being accessed.
- **Good Test Coverage.** Joda-Time has a comprehensive set of developer tests, providing assurance of the library's quality.
- **Complete Documentation.** There is a full User Guide which provides an overview and covers common usage scenarios. The javadoc is extremely detailed and covers the rest of the API.
- **Maturity.** The library has been under active development since 2002. Although it continues to be improved with the addition of new features and bug-fixes, it is a mature and reliable code base.
- **Open Source.**

What should I know in order to use it?

- DateTime class – his main methods & properties
- Duration, Interval and Period
- Chronology & Time Zone
- Interoperation & Formatting
- Local Date & Time
- Usage Examples
- Summary
- Next generation alternative

Joda-Time DateTime

// current time

```
DateTime now = new DateTime();
```

// get fields

```
int y = now.getYear();
```

```
int m = now.getMonthOfYear();
```

```
int d = now.getDayOfMonth();
```

```
int hour = now.getHourOfDay();
```

```
int min = now.getMinuteOfHour();
```

```
int sec = now.getSecondOfMinute();
```

// get week based fields

```
int wy = now.getWeekyear();
```

```
int wk = now.getWeekOfWeekyear();
```

```
int dow = now.getDayOfWeek();
```

Joda-Time Properties

- public [DateTime.Property](#) **dayOfWeek()**
- Property class has many useful methods:
[getAsShortText](#), [getAsShortText](#)(Locale), [getAsString](#) ...
- How to get a property object?
 - `DateTime dt = new DateTime();`
 - `dt. ...`

Joda-Time Properties

- public [DateTime.Property](#) **dayOfWeek()**
- public [DateTime.Property](#) **hourOfDay()**
- public [DateTime.Property](#) **dayOfMonth()**
- public [DateTime.Property](#) **centuryOfEra()**
- Example:

```
System.out.println(dt.dayOfWeek().getAsText());
```

- Monday

Joda-Time Immutability

- **DateTime** instances are immutable
 - API principle: Favour immutability
- Once created, object cannot be changed, like JDK String, Integer or Boolean
- Threadsafe, no need to clone
- Mutator methods return new object

Joda-Time Immutability

```
// current time
DateTime now = new DateTime();

// set the time
DateTime middayToday =
    now.withTime(12, 0, 0, 0).withYear(1998);

// set the zone
DateTime nowZoned = now.withZone(DateTimeZone.UTC);

// add or take away a time period
DateTime yesterday = now.minusDays(1).plusHours(3);
```


Joda-Time Properties

// current time

```
DateTime now = new DateTime();
```

// get the day of week name (Tuesday/Tue/mardi)

```
String dowLong = now.dayOfWeek().getAsText();
```

```
String dowShort = now.dayOfWeek().getAsShortText();
```

```
String fr = now.dayOfWeek().getAsText(Locale.FRENCH);
```

// other queries

```
boolean leap = now.year().isLeap();
```

```
int daysInMonth = now.monthOfYear().getMaximumValue();
```

// mutators

```
DateTime dt1 = now.year().setCopy(1950);
```

```
DateTime dt2 = now.dayOfMonth().roundFloorCopy();
```

```
DateTime dt3 = now.dayOfMonth().roundCeilingCopy();
```

Joda-Time Duration

- A *duration* in Joda-Time represents a duration of time measured in milliseconds.
- The duration is often obtained from an interval
- Durations are a very simple concept, and the implementation is also simple.
- They have no chronology or time zone, and consist solely of the millisecond duration .

Joda-Time Duration

// create

```
Duration duration = new Duration(1000L);
```

// get duration

```
long millis = duration.getMillis();
```

Joda-Time Period

- A *period* in Joda-Time represents a period of time defined in terms of fields, for example:
5 years 3 months 2 days and 12 hours.
- This differs from a duration in that it is inexact in terms of milliseconds.
 - How many milliseconds between February 28th and March 1st? Depends on the year...
- A period can only be resolved to an exact number of milliseconds by specifying the instant (including chronology and time zone) it is relative to .

Joda-Time Period

```
// create  
Period period = new Period(5, 3, 0, 2, 12, 0, 0, 0);  
Period twoHours = Period.hours(2);  
  
// get fields  
int years = period.getYears();  
int days = period.getDays();
```

Joda-Time Interval

An *interval* in Joda-Time represents an interval of time from one instant to another instant. Both instants are fully specified instants in the date/time continuum, complete with time zone.

- Intervals are implemented as *half-open*, which is to say that the start instant is inclusive but the end instant is exclusive. The end is always greater than or equal to the start. Both end-points are restricted to having the same chronology and the same time zone.
- If end time > start time exception occurs

Joda-Time Interval

// create

```
Interval interval1 = new Interval(start, end);
```

```
Interval interval2 = new Interval(start, periodAfter);
```

```
Interval interval3 = new Interval(periodBefore, end);
```

// get fields

```
DateTime start = interval1.getStart();
```

Joda-Time Interval

- An Interval can be converted to a Duration
- Millisecond difference (end - start)

Joda-Time Interval

Convert to Duration

```
DateMidnight start = new DateMidnight(2005, 7, 12);
```

```
DateMidnight end = new DateMidnight(2005, 9, 15);
```

```
Interval interval = new Interval(start, end);
```

```
Duration duration = interval.toDuration();
```

Joda-Time Interval

- An **Interval** can be converted to a **Period**
- Splits into maximum for each field

Joda-Time Interval

```
DateMidnight start = new DateMidnight(2005, 7, 12);  
DateMidnight end = new DateMidnight(2005, 9, 15);  
Interval interval = new Interval(start, end);  
Period period = interval.toPeriod();
```

- Example result is 2 months, 3 days
- Converted to Period

Joda-Time Interval

Convert to Period of Weeks

- What if we want to know the number of complete weeks in the **Interval** ?
- Use a **PeriodType**

Joda-Time Interval

Convert to Period of Weeks

```
DateMidnight start = new DateMidnight(2005, 7, 12);  
DateMidnight end = new DateMidnight(2005, 9, 15);  
Interval interval = new Interval(start, end);  
Period period = interval.toPeriod(PeriodType.weeks());
```

- Example result is 9: weeks

Joda-Time Chronology

- The Joda-Time design is based around the *Chronology* .
- It is a calculation engine that supports the complex rules for a calendar system.
- It encapsulates the field objects, which are used on demand to split the absolute time instant into recognisable calendar fields like 'day-of-week'.
- It is effectively a pluggable calendar system .

Joda-Time Chronology

```
Chronology coptic = Chronology.getCoptic();  
DateTime now = new DateTime(coptic);  
int year = now.getYear();  
  
// year is 1721
```

Joda-Time Time Zone

- The chronology class also supports the time zone functionality.
- This is applied to the underlying chronology via the decorator design pattern.
- The DateTimeZone class provides access to the zones primarily through one factory method, as follows :

DateTimeZone zone =

```
    DateTimeZone.forID ("Europe/London");
```


Joda-Time Time Zone

- In addition to named time zones, Joda-Time also supports fixed time zones.
- The simplest of these is UTC, which is defined as a constant :

```
DateTimeZone zoneUTC = DateTimeZone.UTC;
```

- Other fixed offset time zones can be obtained by a specialize factory method:

```
DateTimeZone myZone = DateTimeZone.forOffsetHours (hours);
```

Joda-Time Time Zone

- Joda-Time provides a default time zone which is used in many operations when a time zone is not specified .
- The value can be accessed and updated via static methods:

`DateTimeZone defaultZone =`

`DateTimeZone.getDefault(); DateTimeZone.setDefault(myZone);`

- JDK implementation codes in time zone rules
- Joda-Time enables you to change them

But All my project works on Date
and Calendar!!!

I use it more than 200 times!

How can I move to using Joda?



Joda-Time Interoperation

- The `DateTime` class has a constructor which takes an `Object` as input.
- In particular this constructor can be passed a JDK `Date`, JDK `Calendar` or JDK `GregorianCalendar`.
- This is one half of the interoperability with the JDK.
- The other half of interoperability with JDK is provided by `DateTime` methods which return JDK objects.

Joda-Time Interoperation

Example:

```
Date date = new Date(1945, 8, 19);  
DateTime dt = new DateTime(date);  
GregorianCalendar c = dt.toGregorianCalendar();  
Date d = dt.toDate();
```

Joda-Time Formatting

- Reading date time information from external sources which have their own custom format is a frequent requirement for applications that have datetime computations.
- Writing to a custom format is also a common requirement .
- Joda time supports these different requirements through a flexible architecture.

Joda-Time Formatting

In general Joda's formatting is:

- Highly powerful formatting support
- Fast
- Thread-safe
- Immutable
- Four format mechanisms
- Access via `DateTimeFormatter`

`DateTimeFormatter f = ... (choose one of mechanisms)`

`String str1 = dateTime.toString(f);`

`String str2 = f.print(dateTime);`

Joda-Time Formatting

Formatting - ISO8601

- Standard format for data interchange
 - 2005-09-20T14:30:00.00+02:00
- Factory is ISODateTimeFormat
- Most common output via toString()

```
DateTime dt = new DateTime(2005, 9, 20, 14, 30, 0, 0);  
DateTimeFormatter f = ISODateTimeFormat.ordinalDate();  
String str = f.print(dt);  
// Output: 2005-263 (263 days past from Jan 1)
```


Joda-Time Formatting

Formatting – Pattern

- Output using a Pattern
 - “dd MMM yyyy”
- Same as SimpleDateFormat
 - some extra letters
- Factory is DateTimeFormat

```
DateTime dt = new DateTime(1978, 10, 3, 14, 30, 0, 0);  
DateTimeFormatter f = DateTimeFormat.forPattern("dd MMM yyyy");  
String str = f.withLocale(Locale.FRENCH).print(dt);
```

```
// Output: 3 oct. 1978
```

Joda-Time Formatting

Formatting – Style

- Output using a Style
 - short, medium, long, full
- Same as DateFormat
- Factory is DateTimeFormat

```
DateTime dt = new DateTime(2005, 9, 20, 14, 30, 0, 0);  
DateTimeFormatter f = DateTimeFormat.longDate();  
String str = f.withLocale(Locale.FRENCH).print(dt);
```

```
// Output: 20 septembre 2005
```

Joda-LocalDate

- This class represents only date without time
- Methods:
 - `getYear()`
 - `getMonthOfYear()`
 - `getDayOfMonth()`
 - `getDayOfWeek()`
 - `getDayOfYear()`

LocalDate – usage example

```
boolean isWorkingDay(LocalDate date) {  
    return date.getDayOfWeek() != DateTimeConstants.FRIDAY &&  
           date.getDayOfWeek() != DateTimeConstants.SATURDAY;  
}
```

Joda-LocalDate

```
    LocalDate nextWorkingDay(LocalDate date) {  
        LocalDate nextDate;  
        int weekDay = date.getDayOfWeek();  
        if (weekDay == DateTimeConstants.THURSDAY) {  
            nextDate = date.plusDays(3);  
        } else if (weekDay == DateTimeConstants.FRIDAY) {  
            nextDate = date.plusDays(2);  
        } else {  
            nextDate = date.plusDays(1);  
        }  
        return nextDate;  
    }
```

Joda-LocalTime

- This class are represent only date without time
- Methods:
 - getHourOfDay()
 - getMinutesOfHour()
 - getSecondsOfMinute()
 - Has properties analogically as DateTime class
 - Has the same mutators methods

JDK vs JODA

In right corner in blue trunks mr. JDK

- JDK datetime classes have serious issues
- Date is mostly deprecated
- Calendar has very odd performance and bugs
- SimpleDateFormat is not Thread-safe
- ...and January = 0, December = 11 !!!

In left corner in red trunks mr.Joda

- Joda-Time is a good alternative
- Simple, yet powerful, API
- Classes are Immutable and Thread-safe
- Consistent performance
- ...and January = 1, December = 12 !!!