

Java Overview

An introduction to the Java Programming Language

Produced by: Eamonn de Leastar (edelestar@wit.ie)
Dr. Siobhan Drohan (sdrohan@wit.ie)



Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
<http://www.wit.ie/>

Unearthing the Excellence in Java



Java

The Good Parts

O'REILLY®

Jim Waldo

Essential Java

⊕ Overview

- ⊕ Introduction
- ⊕ Syntax
- ⊕ Basics
- ⊕ Arrays

⊕ Classes

- ⊕ Classes Structure
- ⊕ Static Members
- ⊕ Commonly used Classes

⊕ Control Statements

- ⊕ Control Statement Types
- ⊕ If, else, switch
- ⊕ For, while, do-while

⊕ Inheritance

- ⊕ Class hierarchies
- ⊕ Method lookup in Java
- ⊕ Use of this and super
- ⊕ Constructors and inheritance
- ⊕ Abstract classes and methods

Interfaces

⊕ Collections

- ⊕ ArrayList
- ⊕ HashMap
- ⊕ Iterator
- ⊕ Vector
- ⊕ Enumeration
- ⊕ Hashtable

⊕ Exceptions

- ⊕ Exception types
- ⊕ Exception Hierarchy
- ⊕ Catching exceptions
- ⊕ Throwing exceptions
- ⊕ Defining exceptions
- Common exceptions and errors

⊕ Streams

- ⊕ Stream types
- ⊕ Character streams
- ⊕ Byte streams
- ⊕ Filter streams
- ⊕ Object Serialization

Overview: Road Map

⊕ Java Introduction

- ⊕ History
- ⊕ Portability
- ⊕ Compiler
- ⊕ Java Virtual Machine
- ⊕ Garbage collection

⊕ Java Syntax

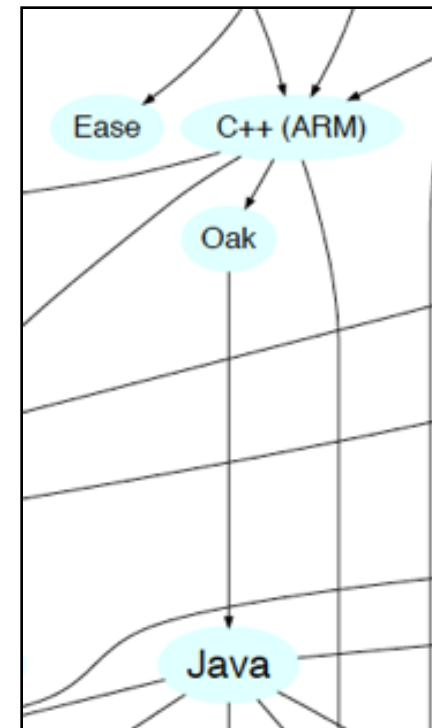
- ⊕ Identifiers
- ⊕ Expressions
- ⊕ Comments

⊕ Java Basics

- ⊕ Java types
- ⊕ Primitives
- ⊕ Objects
- ⊕ Variables
- ⊕ Operators
- ⊕ Identity and equality
- ⊕ Arrays
 - ⊕ What are arrays?
 - ⊕ Creating arrays
 - ⊕ Using arrays

Java History

- ⊕ Originally was called “Oak”.
- ⊕ Was intended to be used in consumer electronics
 - ⊕ Platform independence was one of the requirements
 - ⊕ Based on C++, with influences from other OO languages (Smalltalk, Eiffel...)
- ⊕ Started gaining popularity in 1995
 - ⊕ Renamed to “Java”.
 - ⊕ Was good fit for the Internet applications.



Portability

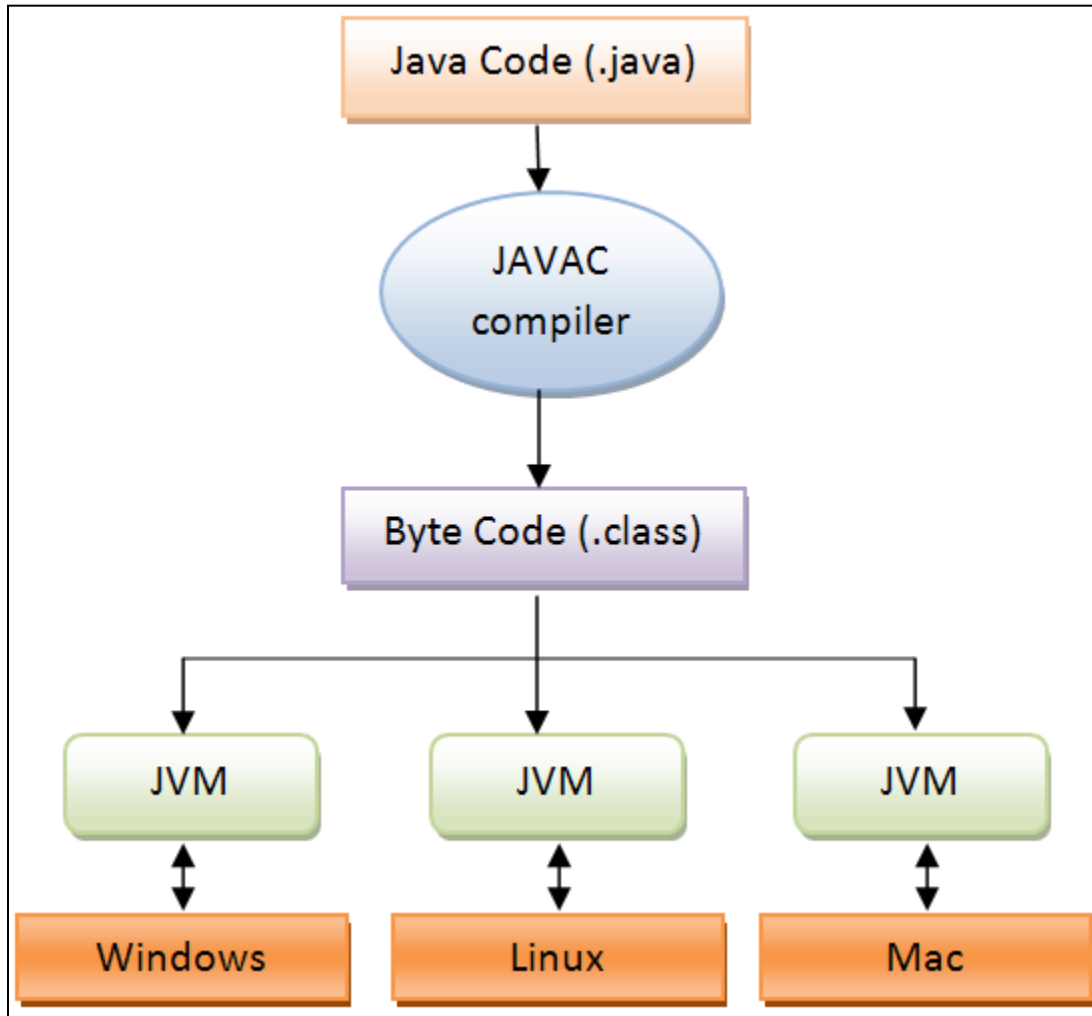
⊕ Java is platform independent language

- ⊕ Java code can run on any platform.
- ⊕ Promotes the idea of writing the code on one platform and running it on any other.

⊕ Java also supports *native methods*

- ⊕ Native methods are written in another language e.g. C, C++.
- ⊕ Native methods are platform specific → don't run on the JVM.
- ⊕ Breaks the idea of platform independence.
- ⊕ Typically used to access/control hardware, interface to system calls or libraries written in other languages, or to improve efficiency in certain tasks.

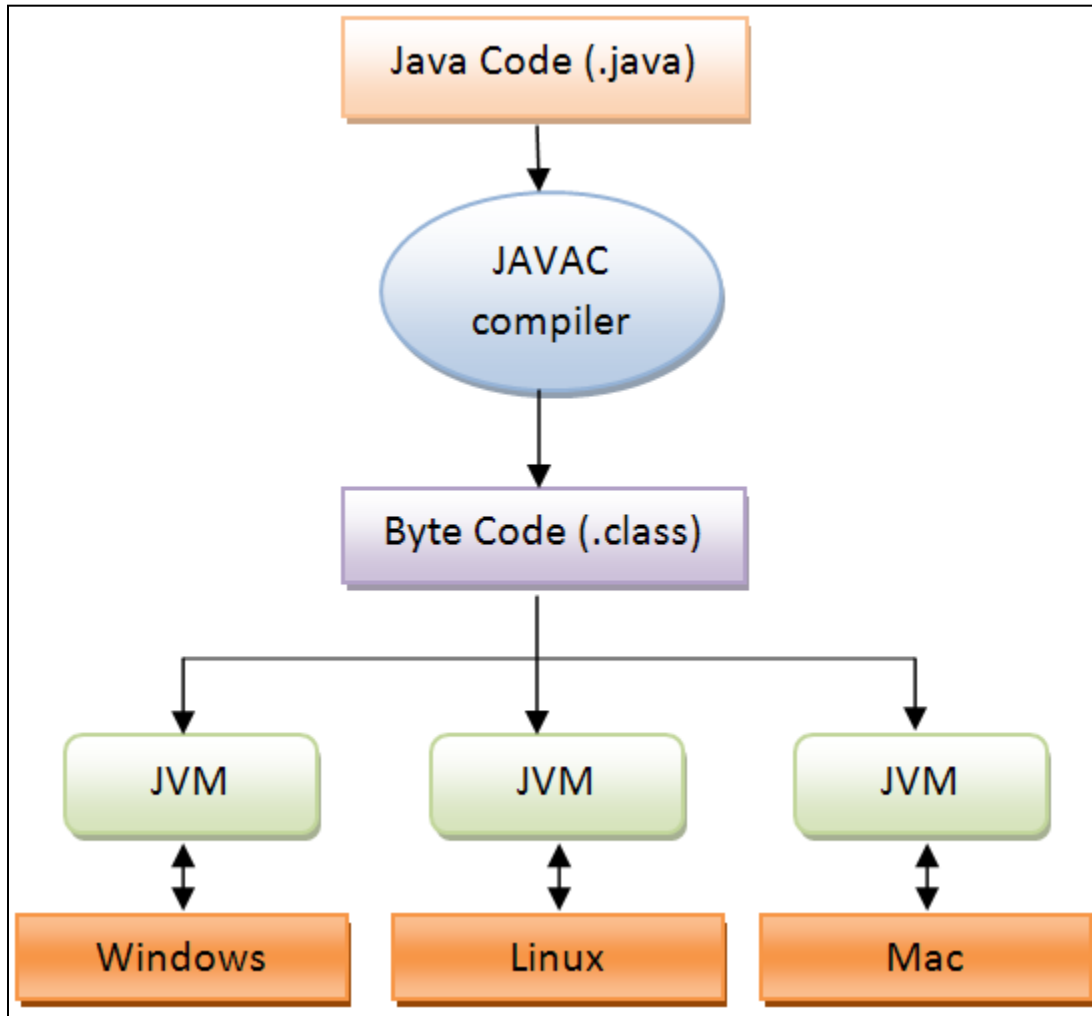
Compiler



⊕ Bytecode:

- ⊕ Can run on any platform with a JVM → platform independent.
- ⊕ Byte code is the same, regardless of platform.
- ⊕ Is not a machine code.
- ⊕ Must be interpreted in the machine code at runtime.

Java Virtual Machine (JVM)

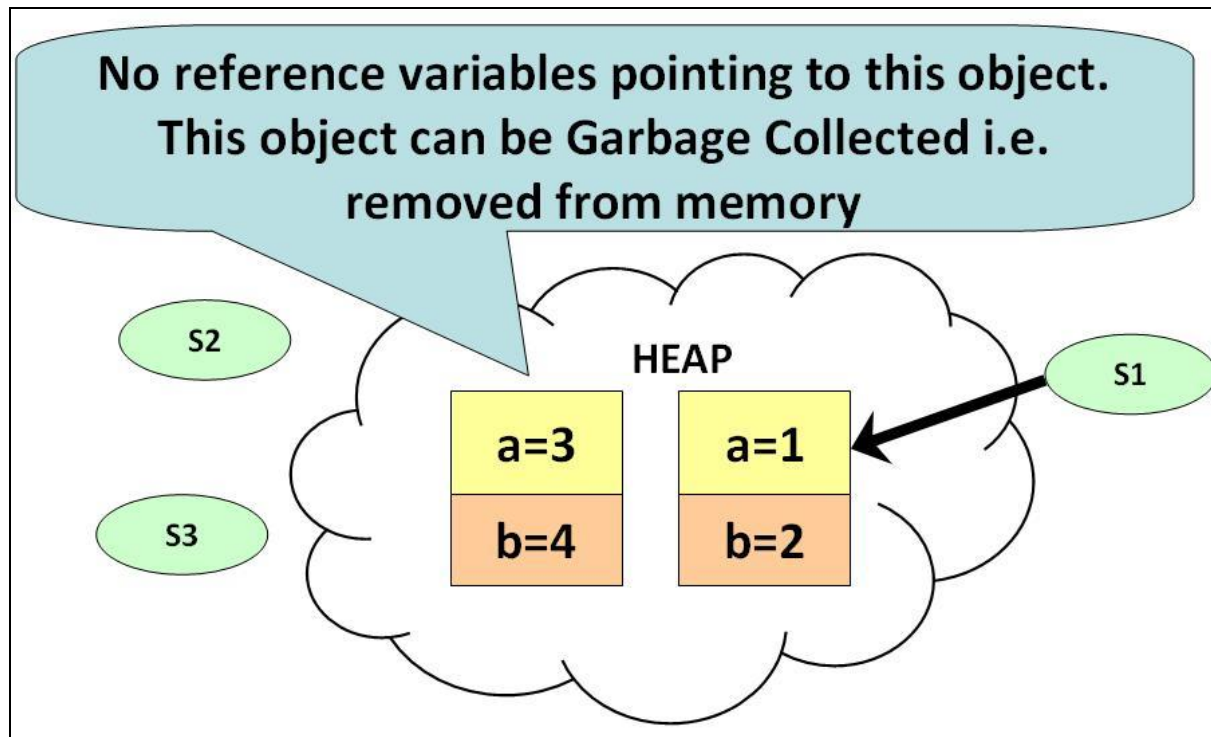


JVM is different for different platforms.

JVM processes bytecode at runtime by translating bytecode into machine code (i.e. interprets it!).

Memory Management

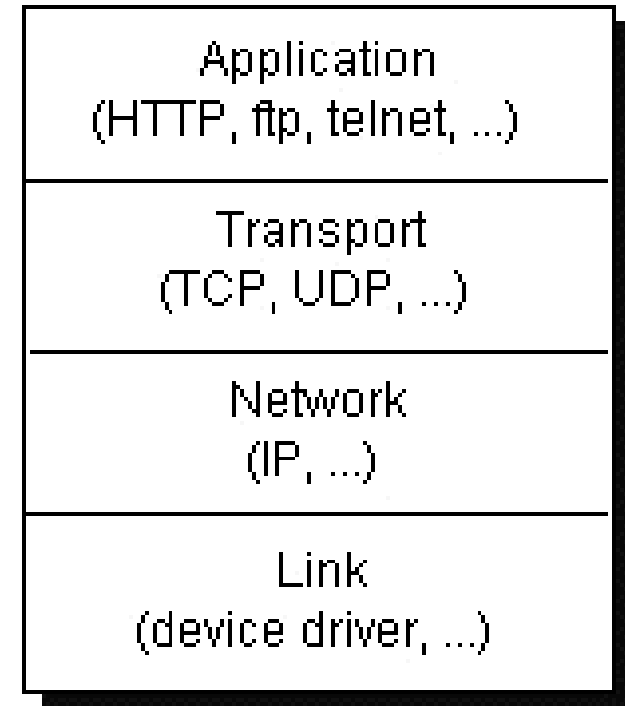
- ⊕ Automatic Garbage Collection is built into Java:
 - ⊕ occurs whenever memory is required
 - ⊕ can be forced programmatically



Distributed Systems (low level)

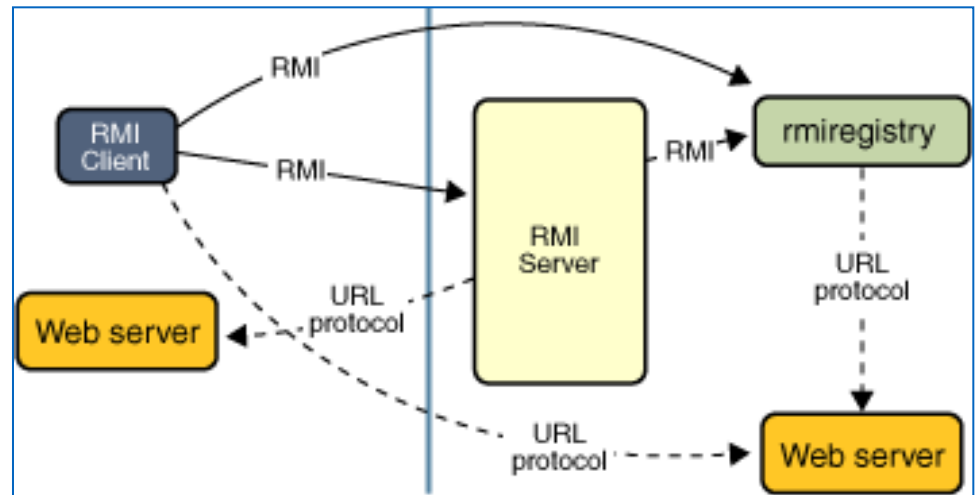
When you write Java programs that communicate over the network, you are programming at the application layer.

Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the `java.net` package. You just need to decide on the protocol.



Distributed Systems (high level)

- Remote Method Invocation (RMI) is Java's distributed protocol, providing for remote communication between programs written in Java.
- Allows an object running in one JVM to invoke methods on an object running in another JVM.
- Use `java.rmi` package.



Concurrency

- ⊕ Java includes support for multithreaded applications
 - ⊕ API for thread management is part of the language
(`java.util.concurrent`)

“A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.”

Overview: Road Map

⊕ Java Introduction

- ⊕ History
- ⊕ Portability
- ⊕ Compiler
- ⊕ Java Virtual Machine
- ⊕ Garbage collection

⊕ Java Syntax

- ⊕ Identifiers
- ⊕ Expressions
- ⊕ Comments

⊕ Java Basics

- ⊕ Java types
- ⊕ Primitives
- ⊕ Objects
- ⊕ Variables
- ⊕ Operators
- ⊕ Identity and equality
- ⊕ Arrays
 - ⊕ What are arrays?
 - ⊕ Creating arrays
 - ⊕ Using arrays

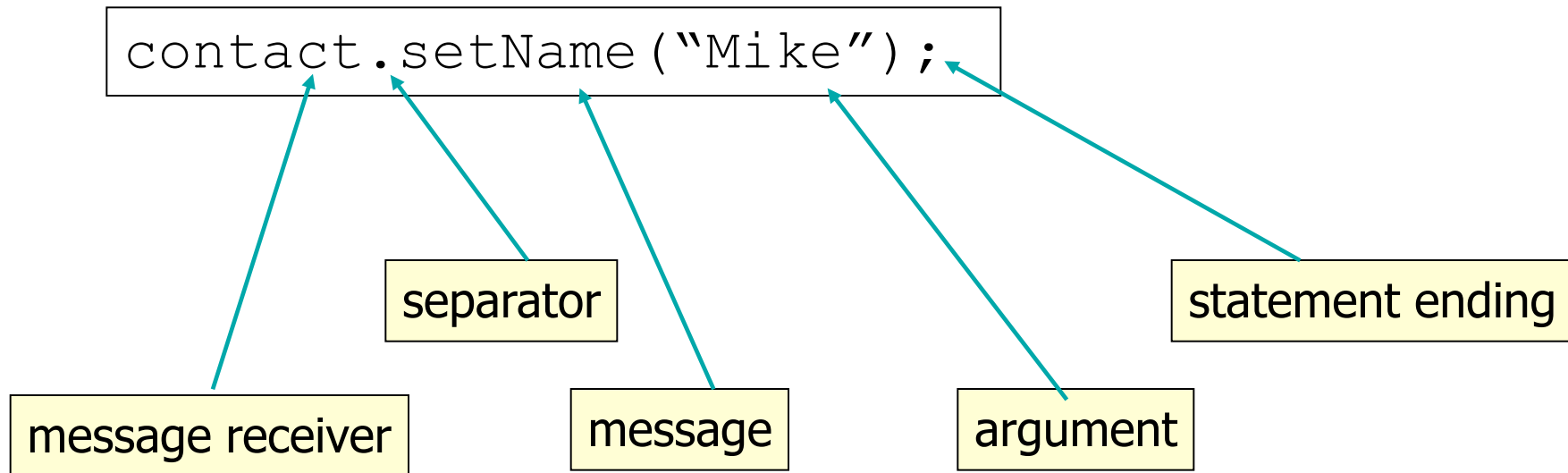
Identifiers

Used for naming classes, interfaces, methods, variables, fields, parameters.

- Are case-sensitive.
- Begin with either:
 - a **letter (preferable)**,
 - the dollar sign "\$", or
 - the underscore character "_".
- Can contain letters, digits, dollar signs, or underscore characters.
- Can be any length you choose.
- Must not be a **keyword or reserved word** e.g. int, while, etc.
- Cannot contain white spaces.

Messages and Objects

- ⊕ Objects send messages to other objects



Expressions

- ⊕ Statements are the basic Java expressions
- ⊕ Semicolon (;) indicates end of a statement

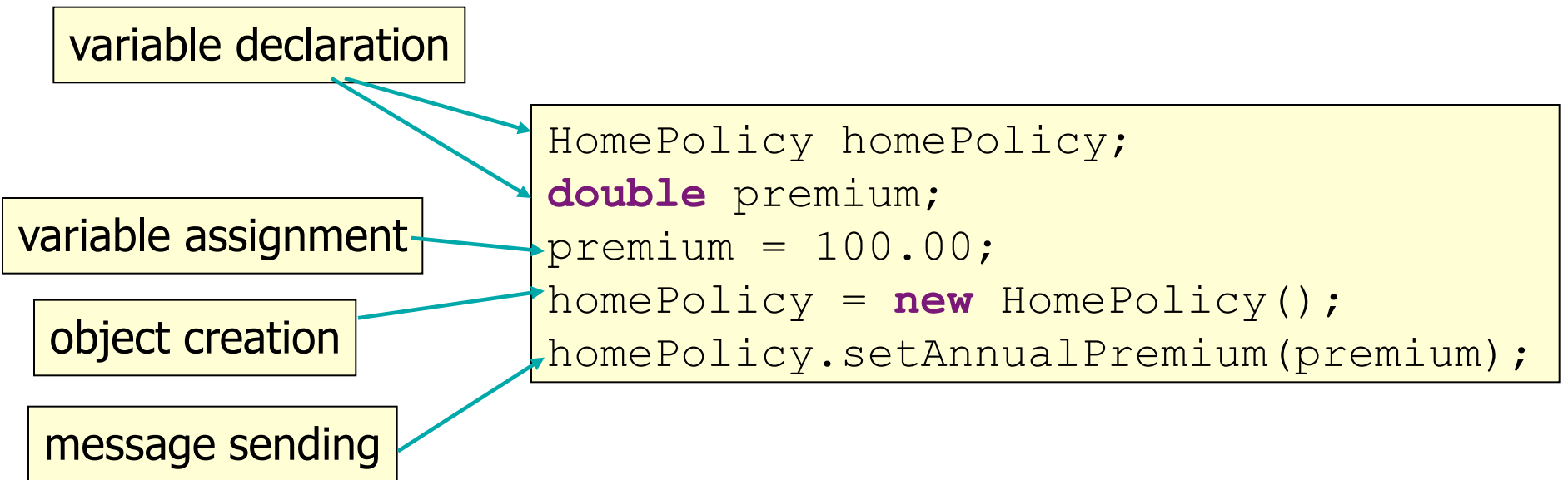
variable declaration

variable assignment

object creation

message sending

```
HomePolicy homePolicy;  
double premium;  
premium = 100.00;  
homePolicy = new HomePolicy();  
homePolicy.setAnnualPremium(premium);
```



Empty Expression

- ⊕ Semicolon on its own in the line
- ⊕ Can be used to indicate *do nothing* scenario in the code

```
;    //this is an empty statement
```

```
for(int i=1; i<3; i++) ;  
    System.out.println(i);
```

- ⊕ We would expect the code to print 1,2 but it prints only 1 because of the empty statement

Comments

⊕ 3 different types of comments in Java:

- ⊕ Single line comment

- ⊕ Starts with // and ends at the end of the line

- ⊕ Multiple line comment

- ⊕ Starts with /* and ends with */

- ⊕ Javadoc comment:

- ⊕ starts with /** and ends with */

- ⊕ used by Javadoc program for generating Java documentation

```
/** Javadoc example comment.  
 * Used for generation of the documentation.  
 */
```

```
/* Multiple line comment.  
 *  
 */
```

```
// Single line comment.
```

Literals

- ⊕ Represent hardcoded values that do not change
- ⊕ Typical example are string literals
 - ⊕ When used compiler creates an instance of String class

```
String one = "One";  
String two = "Two";
```

=

```
String one = new String("One");  
String two = new String("Two");
```

Overview: Road Map

⊕ Java Introduction

- ⊕ History
- ⊕ Portability
- ⊕ Compiler
- ⊕ Java Virtual Machine
- ⊕ Garbage collection

⊕ Java Syntax

- ⊕ Identifiers
- ⊕ Expressions
- ⊕ Comments

⊕ Java Basics

- ⊕ Java types
- ⊕ Primitives
- ⊕ Objects
- ⊕ Variables
- ⊕ Operators
- ⊕ Identity and equality

⊕ Arrays

- ⊕ What are arrays?
- ⊕ Creating arrays
- ⊕ Using arrays

Java and Types

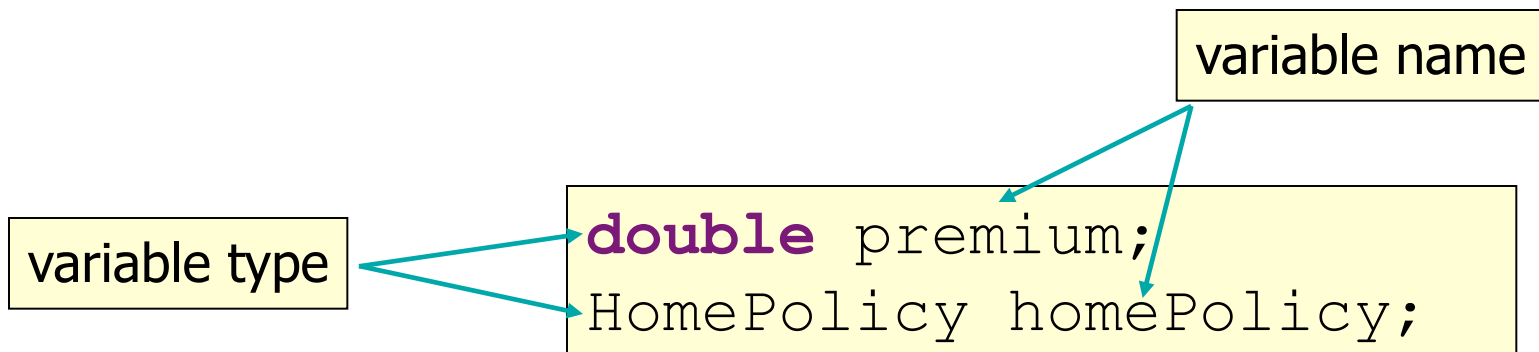
⊕ There are two different types in Java:

⊕ Primitive data type (e.g. premium).

⊕ Reference type (e.g. homePolicy).

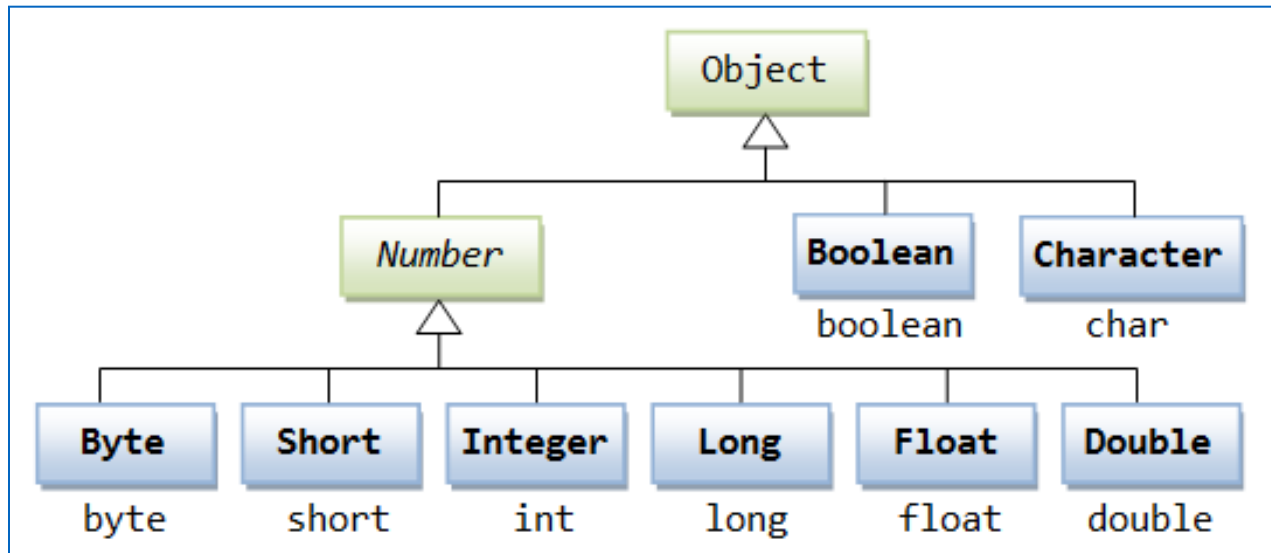
⊕ Java is strongly typed language

⊕ Fields, variables, method parameters and returns must have a type.



Primitives

- ⊕ Primitives represent simple data in Java.
- ⊕ Primitives are not objects in Java
 - ⊕ Messages cannot be sent to primitives.
 - ⊕ Messages can be sent to Wrapper classes that represent primitives (found in `java.lang` package).



Primitive Types

Keyword	Size	Min value	Max value
boolean	true/false		
byte	8-bit	-128	127
short	16-bit	-32768	32767
char	16-bit Unicode		
int	32-bit	-2147483648	2147483647
float	32-bit		
double	64-bit		
long	64-bit	- 9223372036854775808	9223372036854775807

Primitives Operators

Keyword	Description	Keyword	Description	Keyword	Description
+	add	<	lesser	&	and
-	subtract	>	greater	 	or
*	multiple	=	assignment	^	xor
/	divide	>=	greater equal	!	not
%	reminder	<=	less equal	&&	lazy and
(...)	the code within is executed first	==	equals	 	lazy or
++op	increment first	!=	not equal	<<	left bit shift
--op	decrement first	x+=2	x=x+2	>>	right bit shift
op++	increment after	x-=2	x=x-2	>>>	right bit shift with zeros
op--	decrement after	x*=2	x=x*2		

boolean Type

⊕ Commonly used in control statements.

⊕ Consists of two boolean literals:

⊕ true

⊕ false

```
!true           //false
true & true     //true
true | false    //true
```

```
false ^ true    //true
true ^ false    //true
false ^ false   //false
true ^ true     //false
```

XOR: outputs true only when inputs differ (one is true, the other is false)

```
false && true    //false, second operand does not evaluate
true || false   //true, second operand does not evaluate
```

Keyword	Description
!	complement
&	and
	or
^	exclusive or
&&	lazy and
	lazy or

char Type

- ⊕ Represents characters in Java.
- ⊕ Uses 16-bit Unicode for support of internationalization.
- ⊕ Character literals appear in single quotes, and include:
 - ⊕ Typed characters, e.g. `'z'`
 - ⊕ Unicode, e.g. `'\u0040'`, equal to `'@'`
 - ⊕ Escape sequence, e.g. `'\n'`

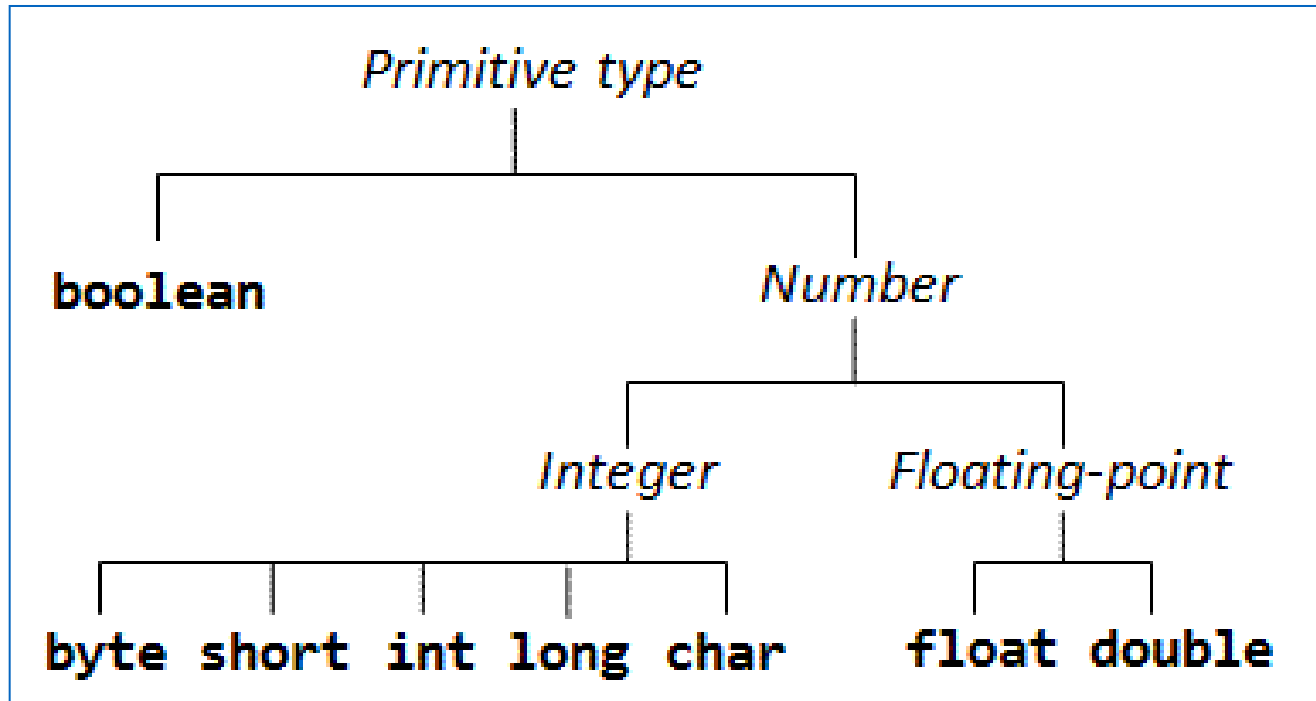
U+0044	D	01000100	LATIN CAPITAL LETTER D
U+0045	E	01000101	LATIN CAPITAL LETTER E
U+0046	F	01000110	LATIN CAPITAL LETTER F
U+0047	G	01000111	LATIN CAPITAL LETTER G
U+0048	H	01001000	LATIN CAPITAL LETTER H
U+0049	I	01001001	LATIN CAPITAL LETTER I
U+004A	J	01001010	LATIN CAPITAL LETTER J
U+004B	K	01001011	LATIN CAPITAL LETTER K
U+004C	L	01001100	LATIN CAPITAL LETTER L

Escape Sequence Characters

⊕ Commonly used with print statements:

Escape sequence	Unicode	Description
<code>\n</code>	<code>\u000A</code>	New line
<code>\t</code>	<code>\u0009</code>	Tab
<code>\b</code>	<code>\u0008</code>	Backspace
<code>\r</code>	<code>\u000D</code>	Return
<code>\f</code>	<code>\u000C</code>	Form feed
<code>\\</code>	<code>\u005C</code>	Backslash
<code>\'</code>	<code>\u0027</code>	Single quote
<code>\"</code>	<code>\u0022</code>	Double quote

Numeric Types



Manipulating Numeric Types

- ⊕ A lesser type is promoted to greater type and then operation is performed

```
12 + 24.56 //int + double = double
```

- ⊕ A greater type cannot be promoted to lesser type
 - ⊕ Assigning double value to int type variable would result in compile error

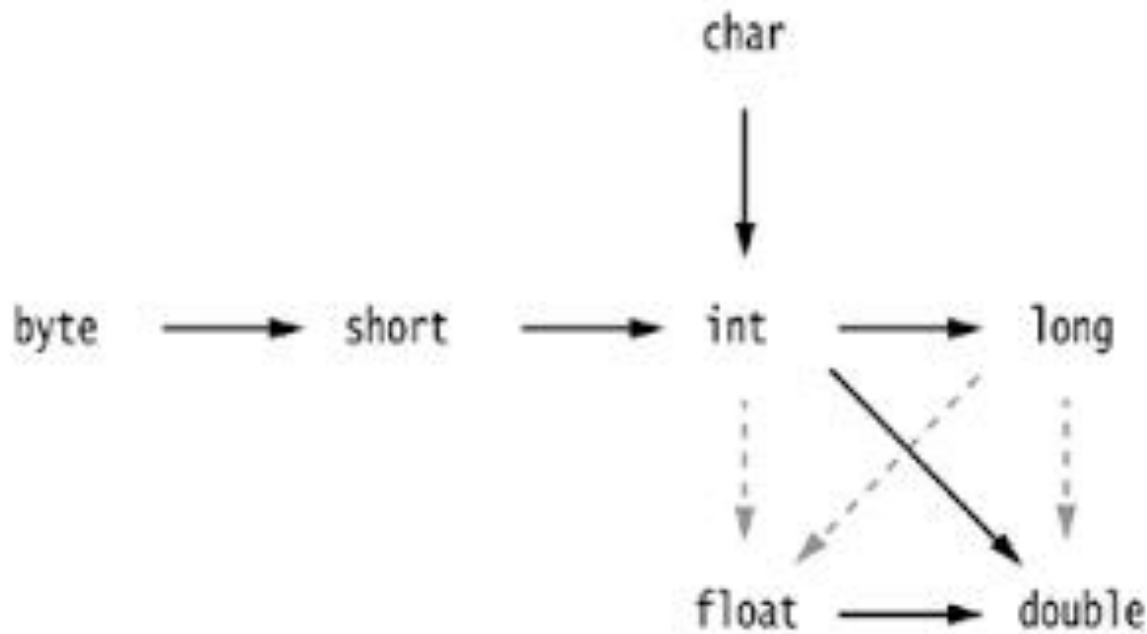
```
int i = 12;  
double d = 23.4;  
i = d;
```



Type mismatch

Manipulating Numeric Types

Legal conversions between numeric types



Hashed arrow: some of the least significant digits may be lost by the conversion.

Type Casting

- ⊕ Values of greater precision cannot be assigned to variables declared as of lower precision types.
- ⊕ Type casting makes primitives to change their type
 - ⊕ Used to assign values of greater precision to variables declared as lower precision
 - ⊕ e.g. it's possible to type cast double to int type

```
int i = 34.5;           //compiler error - type mismatch
int i = (int) 34.5;     //explicit type casting
```

Reference Types

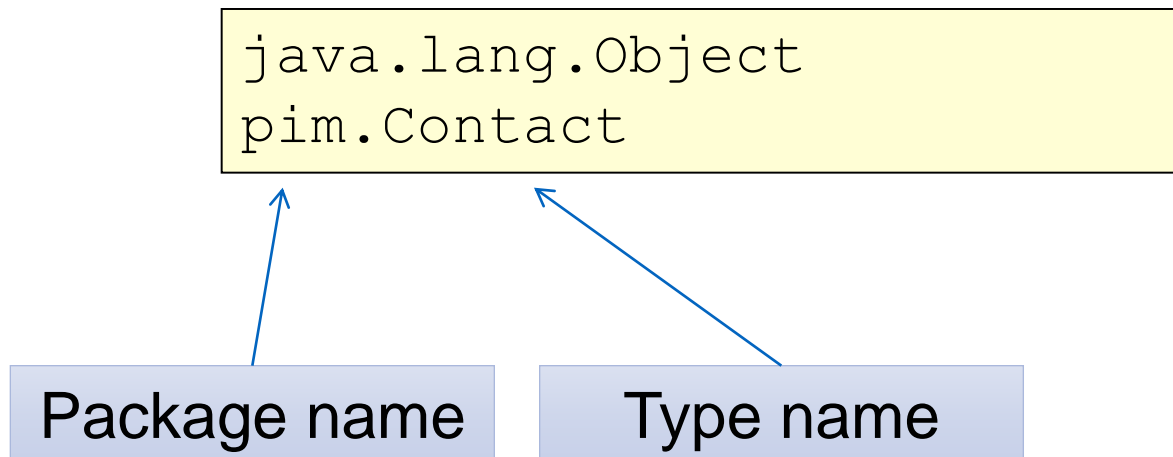
- ⊕ Reference types in Java are class or interface
 - ⊕ also known as object types.

- ⊕ If a variable is declared as a type of class
 - ⊕ An instance of that class can be assigned to it.
 - ⊕ An instance of any subclass of that class can be assigned to it.

- ⊕ If a variable is declared as a type of interface
 - ⊕ An instance of any class that implements the interface can be assigned to it.

Reference Type

- ⊕ Reference type names are uniquely identified by:
 - ⊕ Name of the package where type is defined (class or interface)
 - ⊕ Type name




Object Operators

Keyword	Description
instanceof	object type
!=	not identical
==	identical
=	assignment

Creating Objects in Java

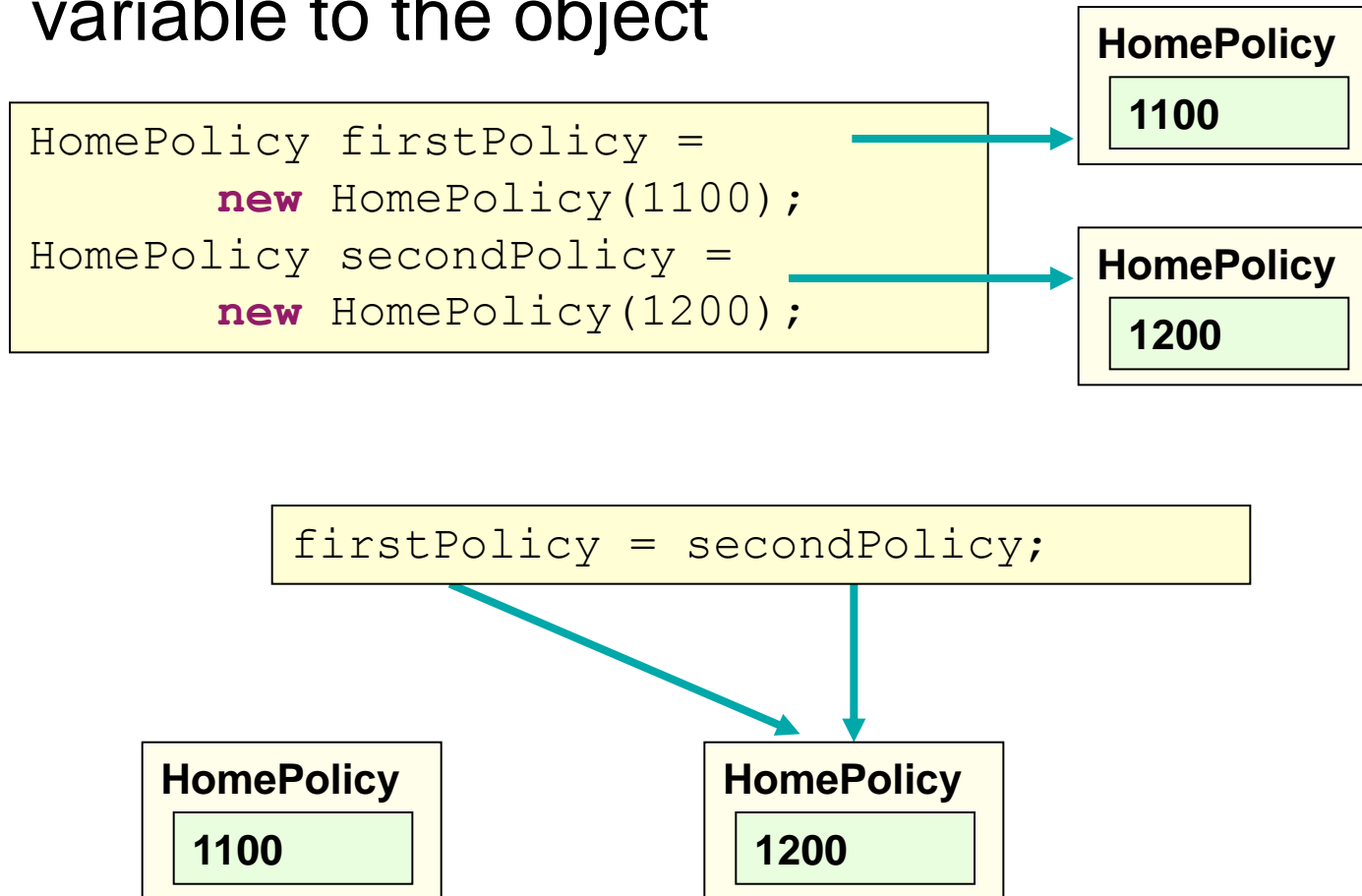
- ⊕ Objects are, in Java, created by using constructors.
- ⊕ Constructors are methods that have same name as the class:
 - ⊕ They may accept arguments mainly used for fields initialization
 - ⊕ If constructor is not defined, the **default constructor** is used.



```
HomePolicy firstPolicy = new HomePolicy();  
HomePolicy secondPolicy = new HomePolicy(1200);
```

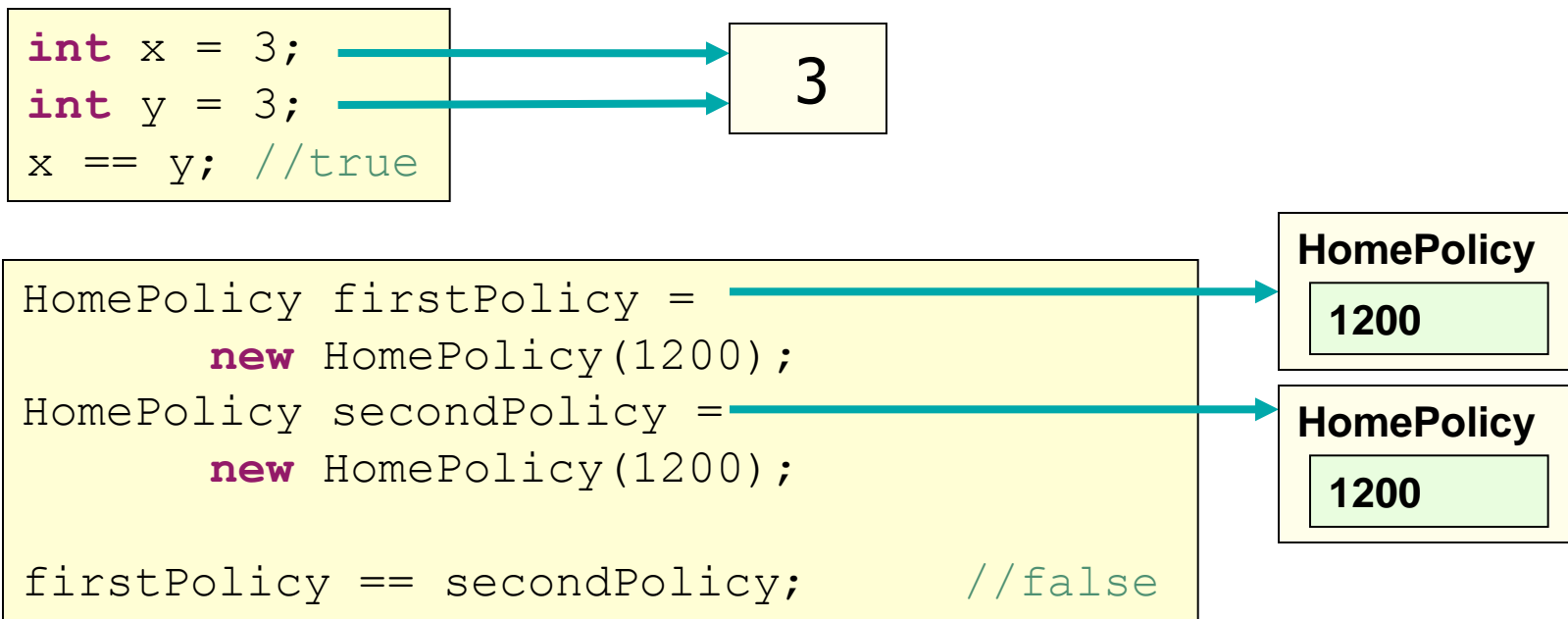
Assignment

- ✦ Assigning an object to a variable binds the variable to the object



Identical Objects...

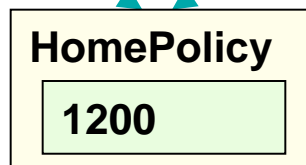
- ⊕ Operand `==` is used for checking if two objects are identical
- ⊕ Objects are identical if they occupy same memory space



...Identical Objects

- ⊕ Variables that reference objects are compared by value
 - ⊕ Objects are identical if their memory addresses are the same
- ⊕ Variables are identical if they refer to exactly same instance of the class

```
HomePolicy firstPolicy = new HomePolicy(1200);  
HomePolicy secondPolicy = firstPolicy;  
firstPolicy == secondPolicy; //true
```



Equal Objects

Determined by implementation of the **equals()** method.

- ⊕ Default implementation is in the Object class (uses identity ==).
- ⊕ Usually overridden in subclasses to provide criteria for equality.

```
HomePolicy firstPolicy =  
    new HomePolicy(1200,1);  
HomePolicy secondPolicy =  
    new HomePolicy(1200,1);  
  
firstPolicy.equals(secondPolicy);
```

null

- ⊕ Used to un-assign an object from a variable
 - ⊕ Object is automatically garbage collected if it does not have a reference.
- ⊕ When a variable of object type is declared it is assigned null as a value.

```
String one = "One";  
one = null;  
one = "1";
```

```
HomePolicy policy;  
policy = new HomePolicy(1200);  
...  
if (policy != null)  
{  
    System.out.println(policy.toString());  
}
```


Overview: Road Map

⊕ Java Introduction

- ⊕ History
- ⊕ Portability
- ⊕ Compiler
- ⊕ Java Virtual Machine
- ⊕ Garbage collection

⊕ Java Syntax

- ⊕ Identifiers
- ⊕ Expressions
- ⊕ Comments

⊕ Java Basics

- ⊕ Java types
- ⊕ Primitives
- ⊕ Objects
- ⊕ Variables
- ⊕ Operators
- ⊕ Identity and equality

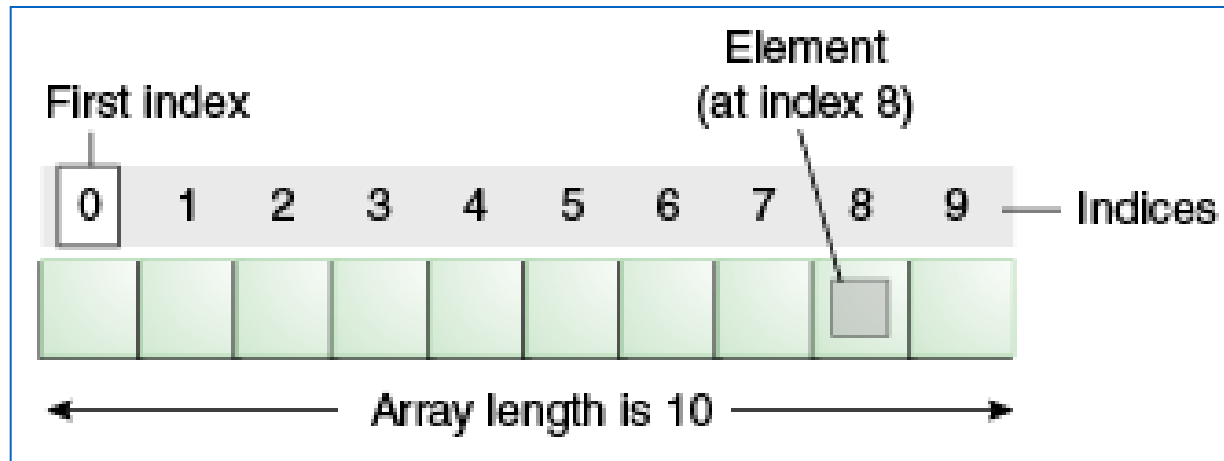
⊕ Arrays

- ⊕ What are arrays?
- ⊕ Creating arrays
- ⊕ Using arrays

What is an Array?

- ⊕ Arrays are basic collections in Java
 - ⊕ They contain elements of the same type
 - ⊕ Elements can either be Java objects or primitives
- ⊕ Arrays are fixed-size sequential collection
 - ⊕ Size is predefined, and arrays cannot grow
- ⊕ Arrays are objects

Array Basics



- ⊕ Arrays are automatically bounds-checked
- ⊕ When accessing elements that are out of bounds, an exception will be thrown e.g.
 - accessing element at index 10 in the above example will throw the exception.

Creating Arrays...

- ⊕ Arrays store objects of specific type
- ⊕ One array cannot store objects of different types, String and int for example.
- ⊕ To define a variable that holds an array, you suffix the type with square brackets []
- ⊕ This indicates that variable references an array

```
int[] arrayOfIntegers;  
String[] arrayOfStrings;
```

or

```
int arrayOfIntegers[];  
String arrayOfStrings[];
```

...Creating Arrays

- ⊕ There are two ways to create an array:
 - ⊕ Explicitly using the keyword `new`
 - ⊕ Using array initializer
- ⊕ When creating an array explicitly its size must be specified:
 - ⊕ This indicates desired number of elements in the array
 - ⊕ Elements in the array are initialized to default values

```
int arrayOfIntegers[];  
arrayOfIntegers = new int[5];
```

Array_INITIALIZER

- ⊕ Used for creating and initializing arrays
- ⊕ Array elements are initialized within the curly brackets

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

- ⊕ Can only be used when declaring variable
- ⊕ Using array initializer in a separate step will result in a compilation error

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

Initializing Arrays

- ⊕ If not using initializer, an array can be initialized by storing elements at proper index

```
int[] arrayOfIntegers;  
arrayOfIntegers = new int[5];  
arrayOfIntegers[0] = 1;  
arrayOfIntegers[1] = 2;  
arrayOfIntegers[2] = 3;  
arrayOfIntegers[3] = 4;  
arrayOfIntegers[4] = 5;
```

Manipulating Arrays

- ⊕ An element of the array is accessed by accessing index at which element is stored

```
int[] arrayOfIntegers = {1,2,3,4,5};  
System.out.println(arrayOfIntegers[2]);
```



Console

3

- ⊕ An array size can be obtained by asking for its length
- ⊕ Used commonly in control statements (loops)

```
int[] arrayOfIntegers = {1,2,3,4,5};  
System.out.println(arrayOfIntegers.length);
```



Console

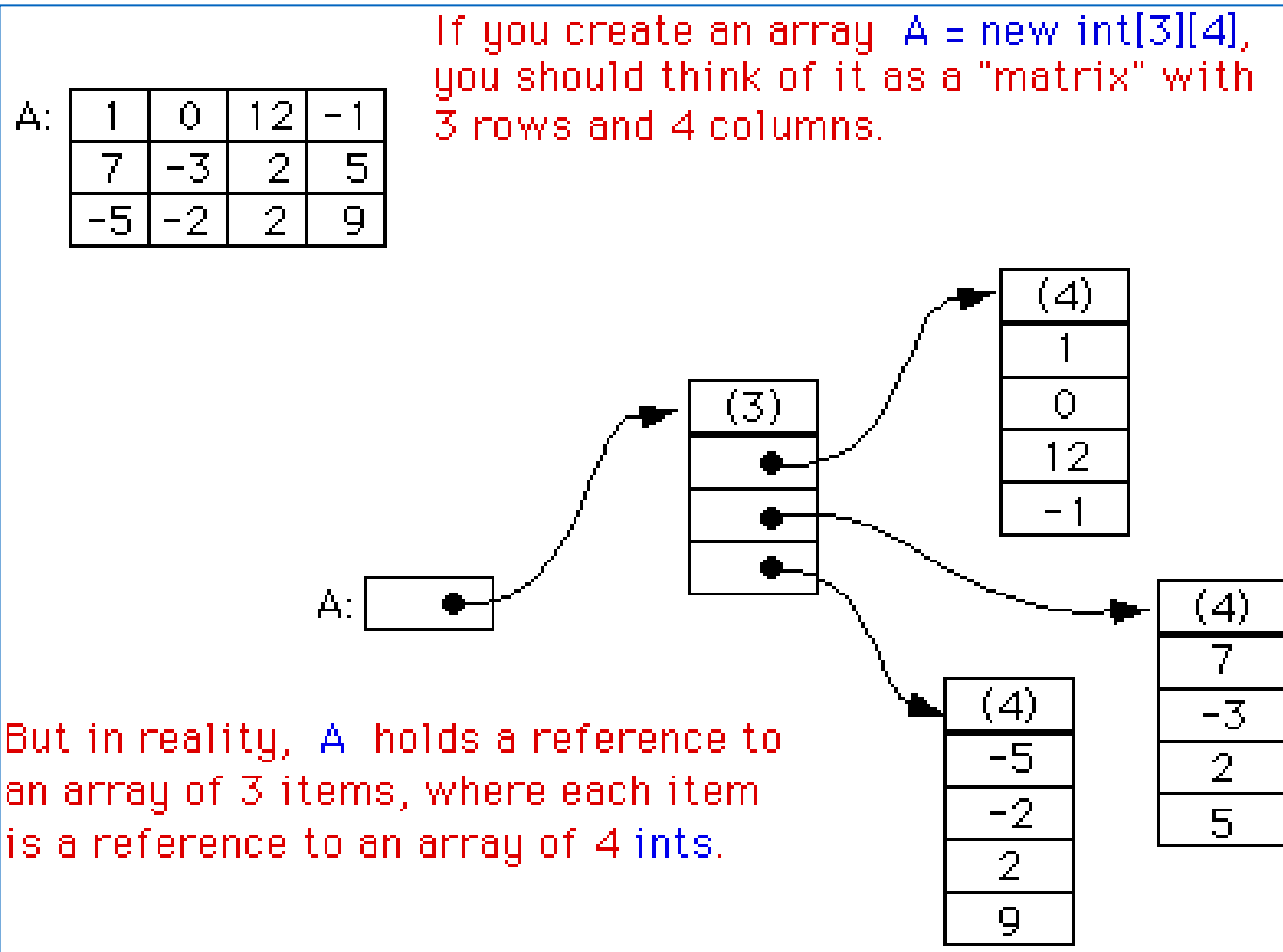
5

Multi-Dimensional Arrays

- ⊕ An array can contain elements of other arrays
 - ⊕ Such an array is known as multi-dimensional array
 - ⊕ There is no limit is number of dimensions
 - ⊕ Arrays can be 2-dimensional, 3-dimensional, and n-dimensional

```
int[][] arrayOfIntegers = new int[2][5];
```

Multi-Dimensional Arrays



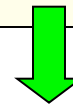
Manipulating Multi-Dimensional Arrays

- ⊕ Multi-dimensional arrays are created like any other arrays
- ⊕ Using the keyword new
- ⊕ Using array initializers

```
int[][] arrayOfIntegers = {{1,2,3,4,5},{6,7,8,9,10}};
```

- ⊕ Elements in multi-dimensional array are also accessed using their indices:

```
int[][] arrayOfIntegers = {{1,2,3,4,5},{6,7,8,9,10}};  
System.out.println(arrayOfIntegers[1][2]);
```



8

Overview: Summary

⊕ Java Introduction

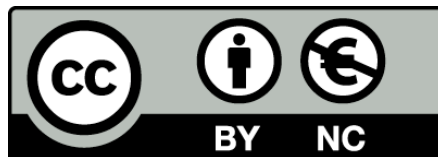
- ⊕ History
- ⊕ Portability
- ⊕ Compiler
- ⊕ Java Virtual Machine
- ⊕ Garbage collection

⊕ Java Syntax

- ⊕ Identifiers
- ⊕ Expressions
- ⊕ Comments

⊕ Java Basics

- ⊕ Java types
- ⊕ Primitives
- ⊕ Objects
- ⊕ Variables
- ⊕ Operators
- ⊕ Identity and equality
- ⊕ Arrays
 - ⊕ What are arrays?
 - ⊕ Creating arrays
 - ⊕ Using arrays



Except where otherwise noted, this content is licensed under a [Creative Commons Attribution-NonCommercial 3.0 License](http://creativecommons.org/licenses/by-nc/3.0/).

For more information, please see <http://creativecommons.org/licenses/by-nc/3.0/>



Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

