# Java overview

History and structure

# Oak – the beginning

Started in 1990 as an internal project of Sun Microsystems to create a

- small
- reliable
- portable
- distributed
- real time

running environment

# Java design aims

Based on analysis of software distribution and running environments:

- *simple – reduced feature set, familiar instructions and syntax*
- *secure – security built into the language and the running environment*
- *high performance – cached interpreter, low priority memory management*
- reliable – check at compile and run time, simple memory model, OOP
- multiplatform – virtual machine executes byte code

# Java

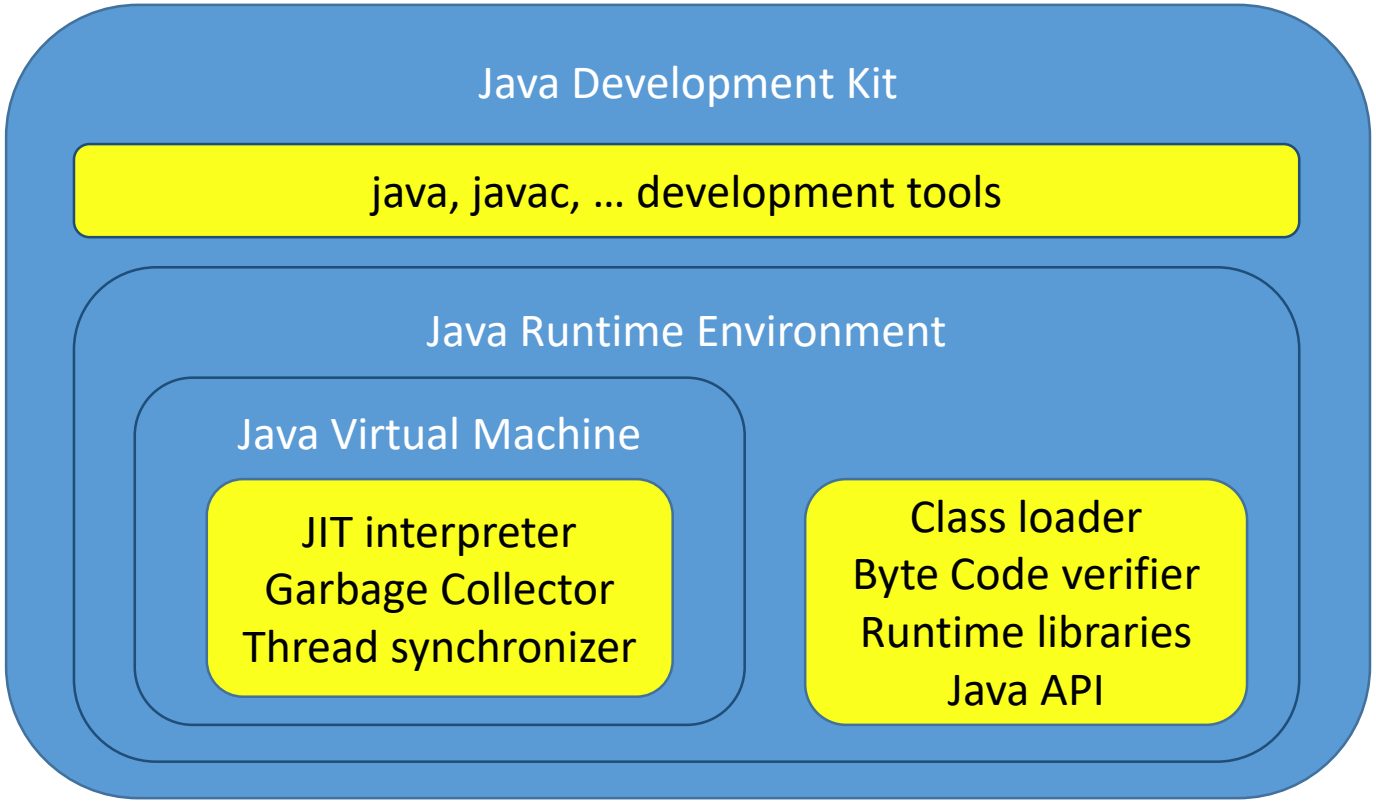
**Advantages**

- Platform independent development
- Platform independent execution
- Unified code optimizer
- Unified memory management

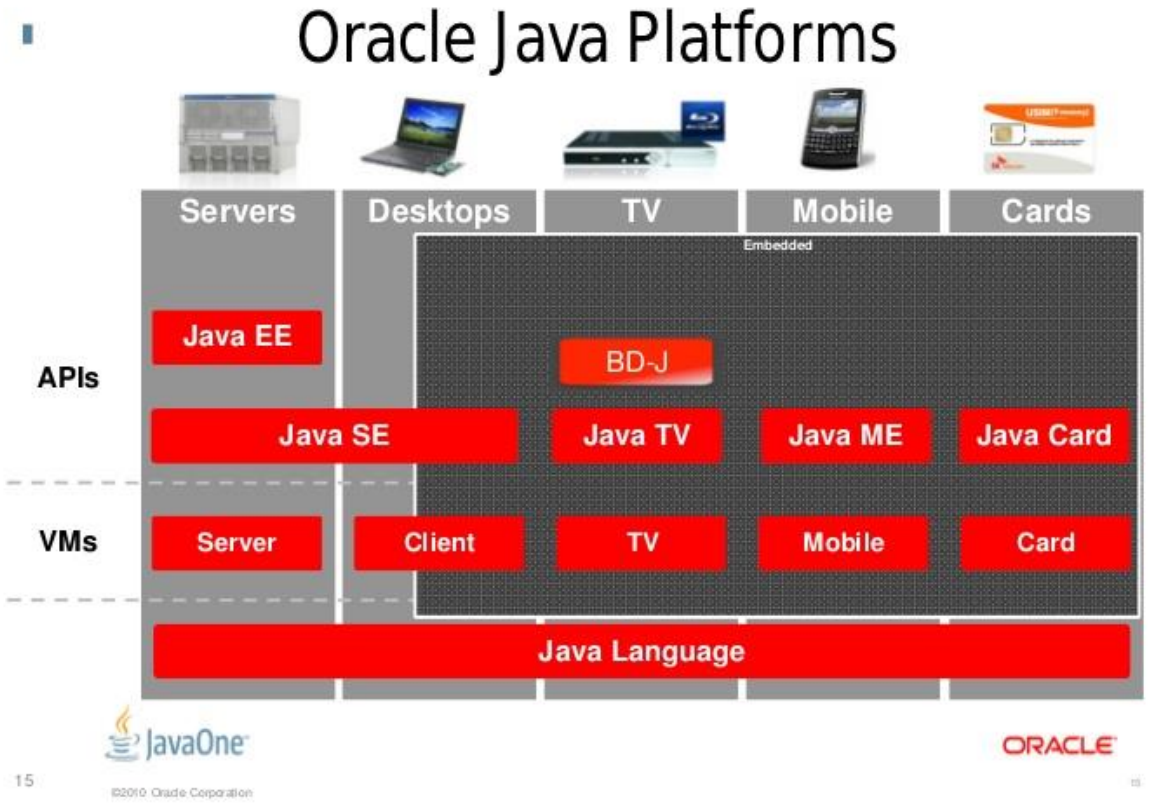
**Disadvantages**

- Requires a running environment
- Slower than native code execution
- Reduced feature set
- Restricted memory access

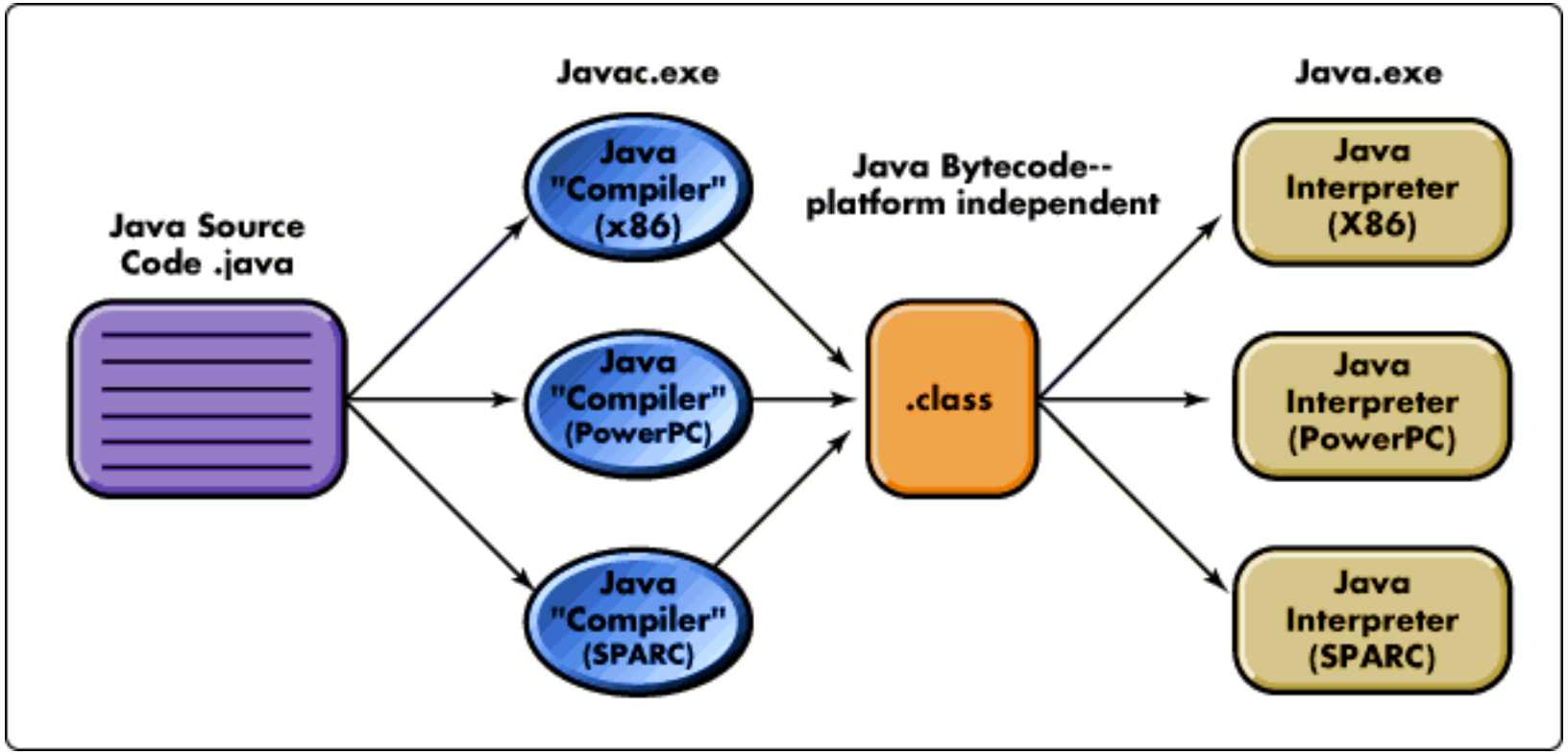
# Java Platform – structure



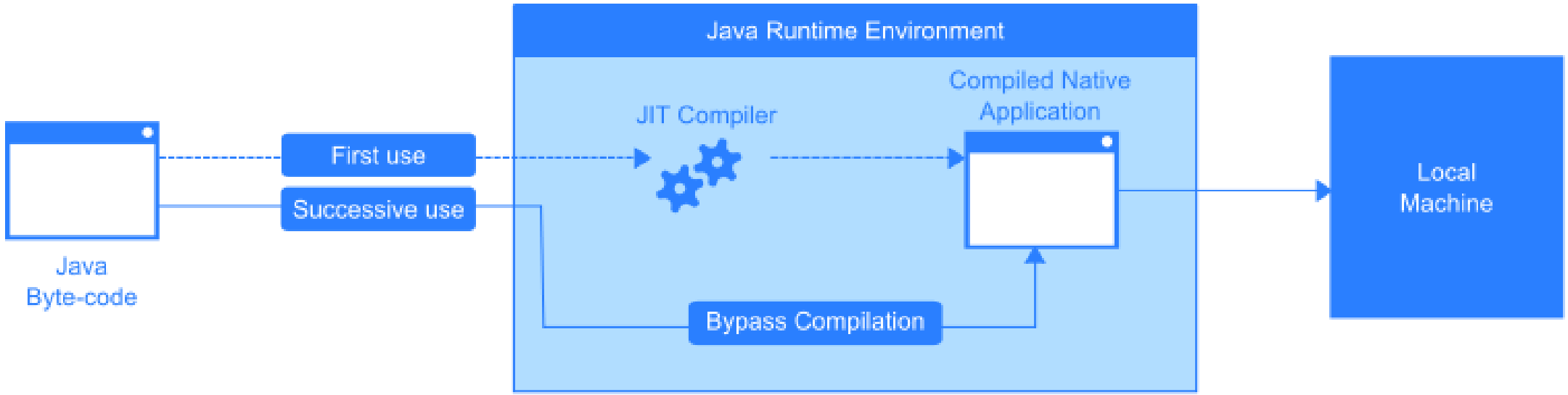
# Java platforms



# Compilation of a Java source code



# Execution of a Java program



# Basic language components – Java

- Változók, típusok, típus átalakítások
- Egyedi típusok: felsorolások, struktúrák
- Tömbök, szöveges típus, konstansok
- Operátorok, operátor overload és precedencia
- Vezérlési szerkezetek
- Kód modulok (függvények), paraméterezésük
- Dinamikus memóriakezelés, adat referenciák
- Függvény pointerek – Metódus referenciák

# Variables

- Like in any other programming languages, they are named memory storage units
- Content is subject of change (if developer does not make otherwise)
- Scope is the declaring code block

# Types

**Primitive types**

- Numeric
  - 8, 16, 32, 64 bit integer and floating point
  - unsigned modifier disappeared
- Character– 16 bit positive integer, 1pc of unicode character
- Boolean – independent type can not be converted to int, values: true/false

# Reference types

- Reference type variables are like C pointers
- Replaces pointer type
- When creating a reference, no allocation is done. That has to be done independently and the reference has to be set to the new allocation
- Compiler tracks the type of the reference and matches to type of allocated area
- Details of design, creation and syntactical rules later…

# Objects

- **Everything** other than primitive type is an object.
- Primitive types can be wrapped into objects
- Basic components of a Java (object oriented) program

- Contains **data** and **operations**
- Based on these above, they have custom type, called **class**

# User defined types

With well designed objects, everything is possible, so Java **does not** contain

- typedef, define, preprocessor
- struct, union
- enum

# Type casts

- Implicit type cast
  - Compatible type to bigger storage space
  - Compatible type to smaller storage space is **NOT ALLOWED**
  - Casting classes later…

```
double length = 5;
int length = 5.0;
//Invalid
```

# Type casts

- Explicit type cast
  - Compatible type to smaller storage space with explicit target type specification

```
double length = 5;
int height =
(int)length/2;
```

# Arrays

- Collection of items if same type
- Not a primitive type → object
- Can be created from any type (even from Object)
- Can be created from array → multi dimensional or nested arrays
- Items accessed by indexer: []
- Can not be indexed like C pointers
- Items have to be initialized one-by-one

# String type

- Not a primitive type → object
- All string data is handled by String objects
- **Not** a character array with terminal
- A String object is immutable, on change, a new object is created
- The compiler handles texts between ”” as String objects

```
String name = ”Tamas”;
```

# Operators

Next to the known arithmetic, logical and relational operations, Java has

- >>> logical right shift
- + (concatenation) of Strings

- No *, &
- Built in operators can not be modified
- Operators can not be created for custom classes (objects)

# Control statements

- Sequence, code block like in C. ”;” and ”{…}”
- `if, for, while, do-while` like in C
- `foreach` – loop through iterable collections without explicit indexing
- Deep nested `break` and `continue`
- **No** `goto`

# Methods, parameters

- Not independent components, parts of objects
- Declaration like in C

```
<return type> <method name>([formal parameters])
{
   <method body>
}
```

# Method parameters

- All parameters are passed by value
- Also reference type variables (like in C)

  Value of reference is the address of referenced region, therefore caller and method work on the same allocated area.

# Dynamic memory management

- Heap is managed by JVM
- Programmer creates references
- On instantiation, JVM allocates memory for the object, sets the reference
- JVM counts references to objects, if an object has no valid reference, it is a subject of removal (garbage collection)

# Method references

Java has multiple technique

- Reference type: `Function<Parameter_types, Return_type> func;`
- Anonymous class – later
- Lambda expression – later
- Method reference: <context>::<method name>

```
func = newCustomer::calculateWage;
```

# Using objects

Java is a strongly typed and object oriented language, therefore developers have to create and use objects.

- Declaration: **`class`** keyword
- Instantiation: **`new`** keyword

Details later…

# Using objects

```
class <class name> {
        [data members]
        [methods]
}

User guest = new User();
Guest.firstName=”Tamas”;
```

```
class User {
    public String firstName;
    public String lastName;

    public String getName() {
        return lastName +” ”+firstName;
    }
}
```

# Console

Without interpretation, use these to access console in terminal applications

- Print data to console:

```
System.console().writer().println(”Hello!”);
```

- Read data from console:

```
String str = System.console().readLine();
```

Details later…

# Console - IDE

Using a IDE, terminal is not accessible, but still able to use developer console.

Without interpretation, use these to access console in terminal applications

- Print data to console:

```
System.out.println(”Hello!”);
```

- Read (scan) data from console:

```
Scanner scanner = new Scanner(System.in);
int i = scanner.nextInt();
```

Details later…