JÖNKÖPING UNIVERSITY

School of Engineering

JAVA

Peter Larsson-Green

Jönköping University

Spring 2020

JAVA DESIGN GOALS

1. Simple, Object Oriented and Familiar.
2. Robust and Secure.
3. Architecture Neutral and Portable
4. High Performance.
5. Interpreted, Threaded, Dynamic.

Syntax similar to C-family.

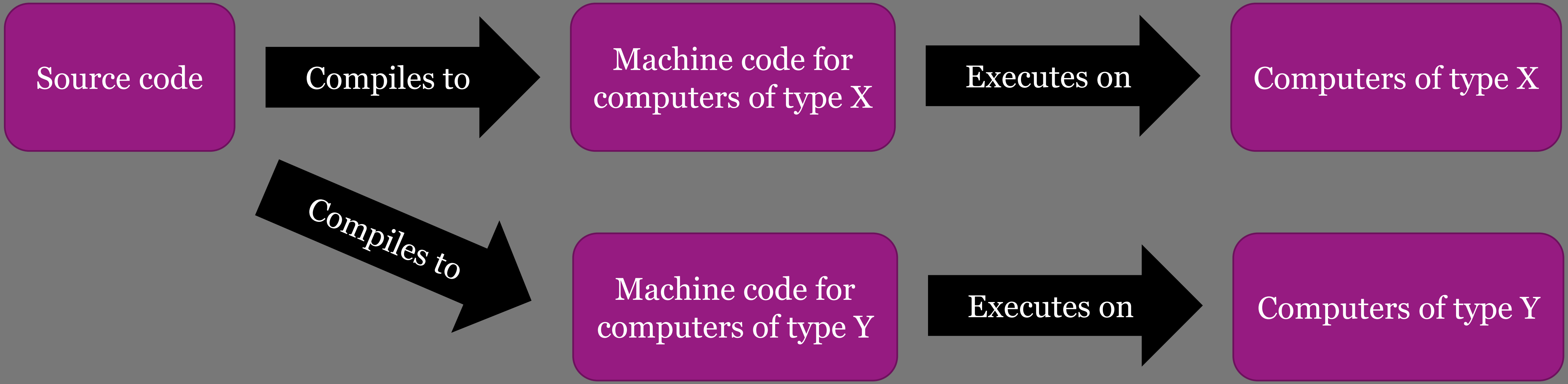
Garbage collection.

Java Virtual Machine.

ARCHITECTURE DEPENDENT

How it works for some languages (C, C++, etcetera).

Drawback: need to compile to multiple targets ☹



Source code

Compiles to

Machine code for computers of type X

Executes on

Computers of type X

Compiles to

Machine code for computers of type Y

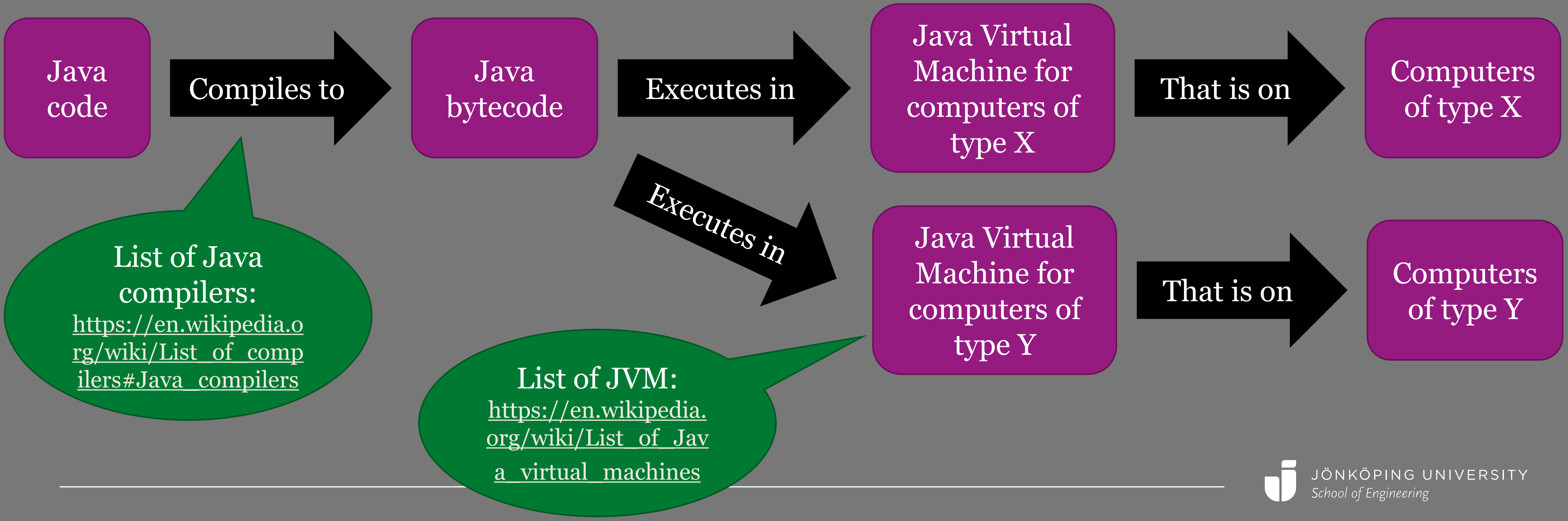
Executes on

Computers of type Y

ARCHITECTURE NEUTRAL

How it works with Java.

Advantage: only compile once ☺
Drawback: Virtual Machines have to be created and installed ☹
Drawback: Virtual Machines are a bit slower ☹



Java code

Compiles to

Java bytecode

Executes in

Java Virtual Machine for computers of type X

That is on

Computers of type X

Executes in

Java Virtual Machine for computers of type Y

That is on

Computers of type Y

List of Java compilers: https://en.wikipedia.org/wiki/List_of_compilers#Java_compilers

List of JVM: https://en.wikipedia.org/wiki/List_of_Java_virtual_machines

GARBAGE COLLECTION

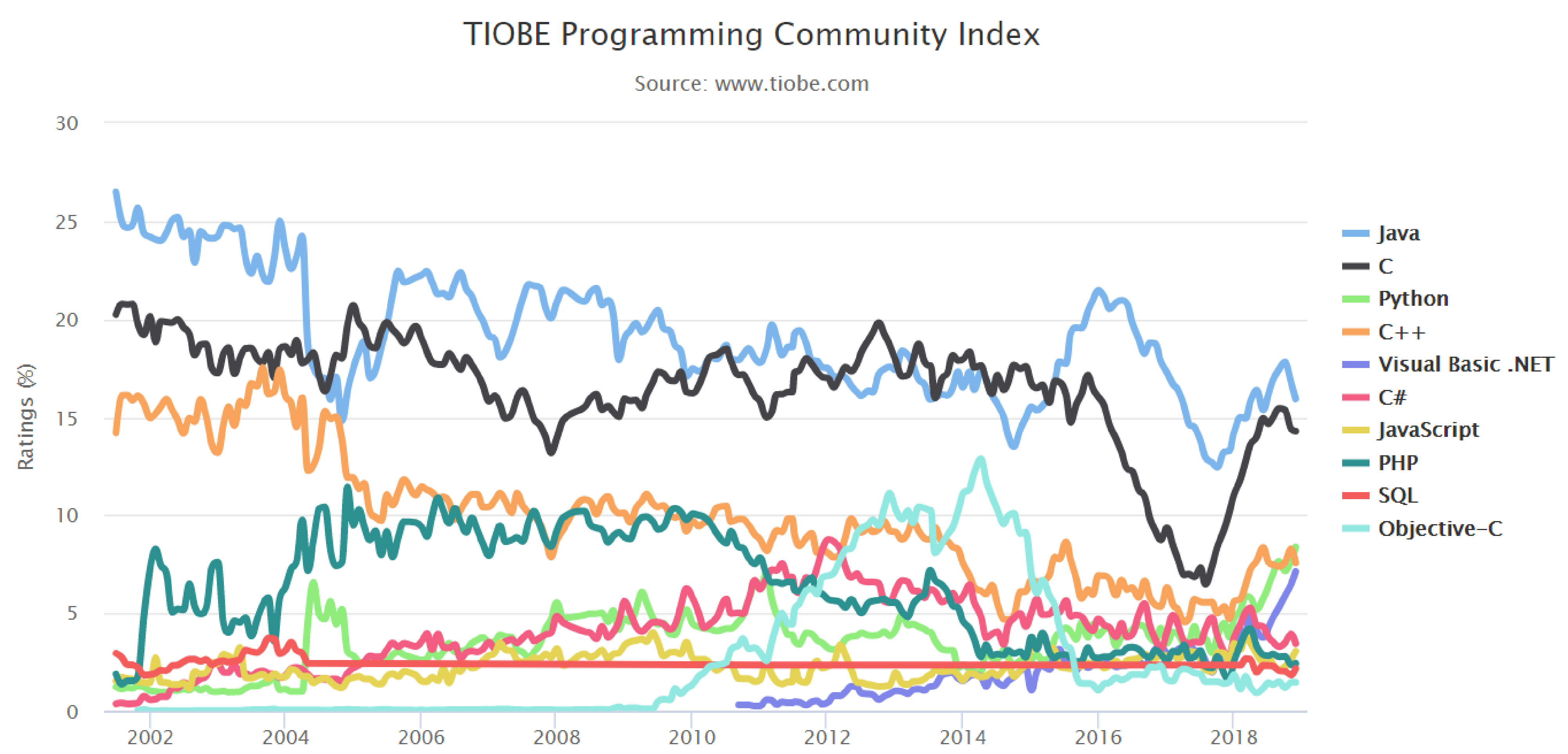
C++

```
for(int i=0; i<100; i++){
  // Create a new object.
  Circle* c = new Circle(5);

  double r = c->getRadius();

  // Free memory.
  delete c;
}
```

Java

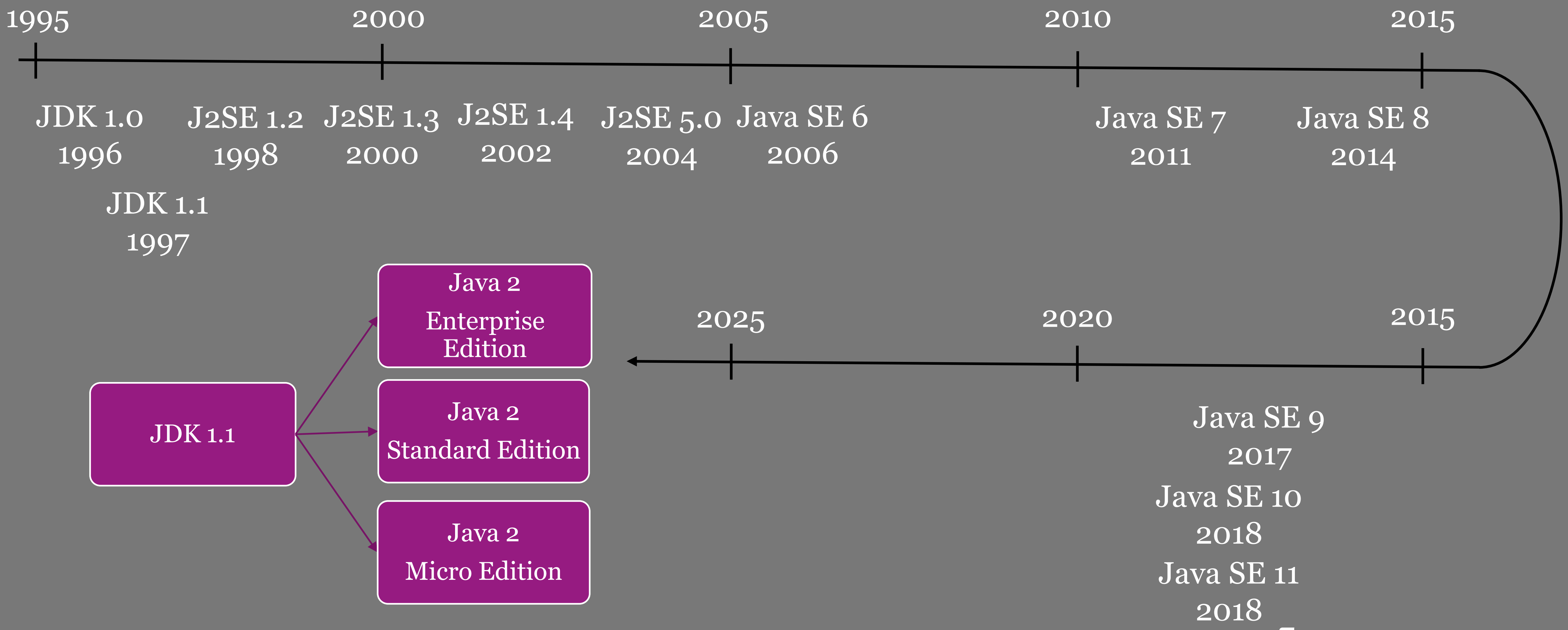
```
for(int i=0; i<100; i++){
  // Create a new object.
  Circle c = new Circle(5);

  double r = c.getRadius();

  // No need to delete it.
  // Handled by the GC.
}
```

THE MOST "POPULAR" LANGUAGES



Source: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

TIMELINE



1995 2000 2005 2010 2015

JDK 1.0
1996

JDK 1.1
1997

J2SE 1.2
1998

J2SE 1.3
2000

J2SE 1.4
2002

J2SE 5.0
2004

Java SE 6
2006

Java SE 7
2011

Java SE 8
2014

2015 2020 2025

Java SE 9
2017

Java SE 10
2018

Java SE 11
2018

JDK 1.1

Java 2
Enterprise Edition

Java 2
Standard Edition

Java 2
Micro Edition

LET'S GET STARTED WITH JAVA

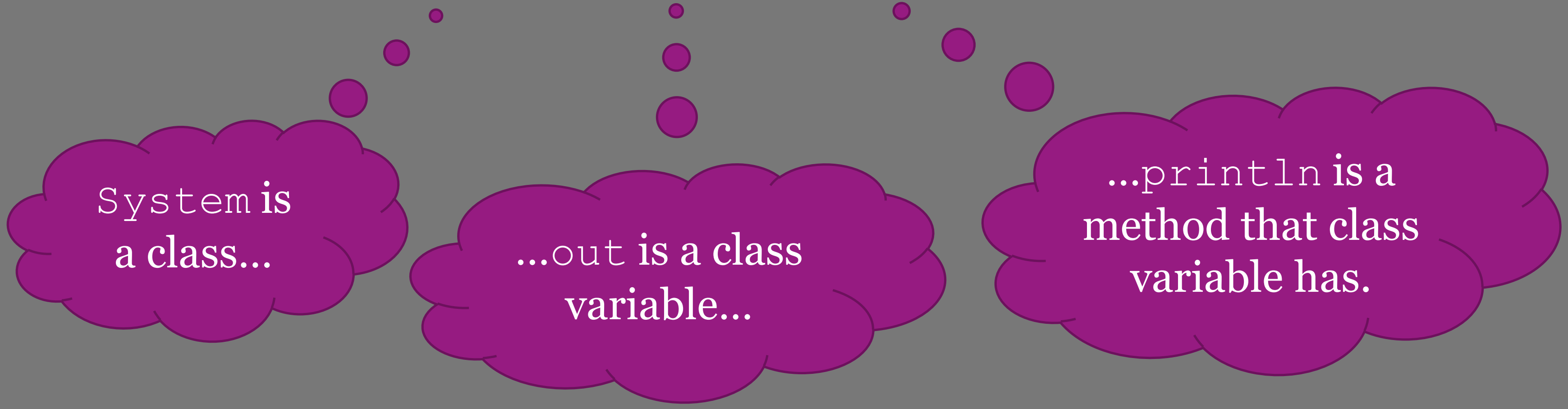
- Files containing Java code should be saved with the `.java` extension.
- Each `.java` file should contain one of the following types: class, interface, enumeration or annotation.
  - The name of the type must match the name of the file.
  - `NamesAreWrittenLikeThis` (CamelCase, first letter capitalized).
- The types are organized in packages.
  - Write the package statement at the top of the file:

    ```
    package se.svensson.sven.package.name;
    ```
- Any class can act as the file containing the main program.
  - The entry point is a class method with the following signature:

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

LET'S GET STARTED WITH JAVA (2)

- Use the *import* statement to import things from a package.
  - `import the.package.Name;` imports `Name` from `the.package`.
  - `import the.package.*;` imports everything from `the.package`.
- Everything in the package `java.lang` is imported by default.
- Use `System.out.println(...)` to write to the console.



`System` is a class...

...`out` is a class variable...

...`println` is a method that class variable has.

HELLO WORLD

```java
package se.ju.larpet.testprogram;

public class MyProgram{
  public static void main(String[] args){
    // Here we put our main program code.
    System.out.println("Hello World!");
  }
}
```

MyProgram.java

HOW TO RUN THE CODE

- Compile `.java` files to `.class` files (`MyClass.java` → `MyClass.class`).
  - The `javac` command is used for this.
- Run the `.class` file with the `main` method in a JVM.
  - The `java` command is used for this.
- Multiple classes are better packaged into a *jar* (Java ARchive) file.
  - Jar files are ZIP files with `.jar` extension.
  - The `jar` command is used for this.
  - Libraries/Programs are distributed as jar files.
    - To run a jar file: `java -jar the-jar-file.jar`

EXAMPLE

LOCAL VARIABLES

Can be used inside constructors and methods.

```
<datatype> variableName = <expression>;
```

- Naming convention: `likeThis` (camelCase, first letter lowercase).
- Can later be assigned a new value.

```
variableName = <expression>
```

PRIMITIVE DATATYPES

Datatype	Size	Min value	Max value	Default value (for fields)
byte	8 bits	`-128`	`127`	`0`
short	16 bits	`-32 768`	`32 767`	`0`
int	32 bits	`-2 147 483 648`	`2 147 483 647`	`0`
long	64 bits	-2^{63}	$2^{63}-1$	`0L`
float	32 bits			`0.0f`
double	64 bits			`0.0`
char	16 bits			`'\u0000'`
boolean				`false`

NUMERICAL BINARY OPERATORS

Operator	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%

- If one operand is `double`
  → convert the other to `double`.
- Otherwise, if one operand is `float`
  → convert the other to `float`.
- Otherwise, if one operand is `long`
  → convert the other to `long`.
- Otherwise, convert both to `int`s.

The outcome has the same datatype as the (possibly converted) operands.

- int / int → int
- int / 0 → throws `ArithmeticException`
- decimal number / 0.0 → `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY`
- short + short → int

RELATIONAL OPERATORS

Operation	Symbol
Less than	<
Greater than	>
Equal to	==
Not equal to	!=
Less than or equal to	<=
Greater than or equal to	>=

- If one operand is `double`
  → convert the other to `double`.
- Otherwise, if one operand is `float`
  → convert the other to `float`.
- Otherwise, if one operand is `long`
  → convert the other to `long`.
- Otherwise, convert both to `ints`.

LOGICAL OPERATORS

- Only works on Booleans!
- Most common logical operators:
  - Not:
    ```
    !<expression>
    ```
  - Or:
    ```
    <expression> || <expression>
    ```
  - And:
    ```
    <expression> && <expression>
    ```

Are lazy!

- More operators exist, e.g.:
  - Bitwise or: `<expression> | <expression>`
  - Bitwise and: `<expression> & <expression>`

THE IF STATEMENT

```
if(<expression>){
   <statements>
}else if(<expression>){
   <statements>
}else{
   <statements>
}
```

THE (DO) WHILE LOOP

```
while(<expression>){
   <statements>
}
```

```
do{
   <statements>
}while(<expression>);
```

THE FOR LOOP

```
for(<initialization>; <condition>; <change>){
   <statements>
}
```

Typically use the variable `i`, storing an `int`.

ARRAYS

```
<datatype>[] variableName = new <datatype>[<expression>];
```

- All elements must be of `<datatype>`.
- The first index is `0`.
- Retrieve element at position `<index>`:

  `variableName[<index>]`
- Assign `<newValue>` to element at position `<index>`:

  `variableName[<index>] = <newValue>`
- Number of elements in the array:

  `variableName.length`

ARRAYS EXAMPLE

```
public class NumbersProgram{
  public static void main(String[] args){
    int[] numbers = new int[3];
    numbers[0] = 3;
    numbers[1] = 1;
    numbers[2] = 4;
    int sum = numbers[0] + numbers[1] + numbers[2];




    System.out.println(sum);
  }
}
```

ARRAYS EXAMPLE

```java
public class NumbersProgram{
  public static void main(String[] args){
    int[] numbers = new int[3];
    numbers[0] = 3;
    numbers[1] = 1;
    numbers[2] = 4;
    int sum = 0;
    for(int i=0; i<numbers.length; i++){
      sum += numbers[i];
    }
    System.out.println(sum);
  }
}
```

INITIALIZING ARRAYS

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

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

THE ENHANCED FOR LOOP

- Iterates through arrays and collections.
  - Collection = class implementing the interface `Iterable<T>`.
- No index.

```
for(<datatype> variableName : collection){
   <statements>
}
```

ARRAYS EXAMPLE

```java
public class NumbersProgram{
  public static void main(String[] args){
    int[] numbers = new int[3];
    numbers[0] = 3;
    numbers[1] = 1;
    numbers[2] = 4;
    int sum = 0;
    for(int i=0; i<numbers.length; i++){
      sum += numbers[i];
    }
    System.out.println(sum);
  }
}
```

```java
public class NumbersProgram{
  public static void main(String[] args){
    int[] numbers = {3, 1, 4};



    int sum = 0;
    for(int number : numbers){
      sum += number;
    }
    System.out.println(sum);
  }
}
```

CLASS VISIBILITY

Sets restriction on who may use the class.

```
package sample.package;


<visibility> class MyClassName{
   // Code for the class...
}
```

<visibility>	Accessible
public	Everywhere.
(default)	Same package.

MEMBER VISIBILITY

Used for fields, constructors, methods and nested types.

```
package sample.package;

public class ClassName{
  <visibility> <datatype> fieldName;
  <visibility> ClassName(){}
  <visibility> <datatype> methodName(){}
  <visibility> class NestedClassName{
  }
}
```

`<visibility>`	**Accessible**
`public`	Everywhere.
`protected`	The package + subclasses in other packages.
(default)	The package.
`private`	The class itself only.

A COUNTER EXAMPLE

```
public class Counter{
  private int value;
  public Counter(int startValue){
    value = startValue;
  }
  public void increment(int amount){
    value += amount;
  }
  public int getValue(){
    return value;
  }
}
```

```
public class TestCounterProgram{
  public static void main(String[] args){
    Counter c = new Counter(5);
    c.increment(3);
    System.out.println(c.getValue());
  }
}
```

WHERE'S `THIS`?

- In constructors, `this` refers to the object being created.
- In methods, `this` refers to the object calling the method.
- In most cases it's not needed.
  - If a name can't refer to anything else but a member on `this`, Java will use that member.

A COUNTER EXAMPLE

```java
public class Counter{
  private int value;
  public Counter(int startValue){
    value = startValue;
  }
  public void increment(int amount){
    value += amount;
  }
  public int getValue(){
    return value;
  }
}
```

```java
public class Counter{
  private int value;
  public Counter(int value){
    this.value = value;
  }
  public void increment(int amount){
    value += amount;
  }
  public int getValue(){
    return value;
  }
}
```

METHOD OVERLOADING

Methods can have the same name.

- Their parameters must be different.

The same goes for constructors.

- `this(<arguments>)` calls other constructors (must be first statement).

```
public class MyClass{
  public MyClass(int aNumber){
    // Constructor with one int parameter.
  }
  public MyClass(){
    this(64);
  }
  public static void main(String[] args){
    MyClass object1 = new MyClass(1);
    MyClass object2 = new MyClass();
  }
}
```

CONSTRUCTORS

- All classes must have at least one.
- If you write none, Java will add one for you.
  - Called the default constructor.
  - Got no parameters.

INHERITANCE

A class always inherit members from *one* other class.

- Default super class: `Object`
- You specify the super class:
  ```
  public class MyClass extends TheSuperClass{ ... }
  ```
- The keyword `super` refers to the super class.
  - In constructors, call a constructor in the super class:
    ```
    super(<arguments>)
    ```
  - In methods, call a method in the super class:
    ```
    super.methodName(<arguments>)
    ```
- Constructors are not explicitly inherited.
- One constructor in the super class must be called.
  - If you don't call it, Java will do it for you (the default constructor).

INHERITANCE EXAMPLE

```java
public class ClassA{
  private int theNumber;
  public ClassA(int aNumber){
    theNumber = aNumber;
  }
  public int getNumber(int factor){
    return theNumber*factor;
  }
}
```

```java
public class ClassB extends ClassA{
  public ClassB(){
    super(42);
  }
  @Override
  public int getNumber(int term){
    return super.getNumber(2)+term;
  }
}
```

Dear compiler,
We are overriding this
method on purpose.

ABSTRACT METHODS & CLASSES

The keyword `abstract` makes classes and methods abstract.

- Example abstract method:
  ```
  public abstract void methodName();
  ```
- Example abstract class:
  ```
  public abstract class ClassName{ ... }
  ```
  - Can't be instantiated.
  - Must be abstract if it has at least one abstract method.

INTERFACES

- Java doesn't support multiple inheritance.
  - Instead, Java has interfaces.
- An interface is a collection of public abstract methods.
  - And static methods, and default methods, ...
- A class can implement multiple different interfaces:

```
public class MyClass implements InterfaceA, InterfaceB{ ... }
```

INTERFACE EXAMPLE

```java
public interface SimpleCalculator{
  void add(double number);
}
```

```java
public interface ComplexCalculator{
  void multiply(double number);
}
```

```java
public class Calculator implements
SimpleCalculator, ComplexCalculator{
  private double memory;
  public Calculator(){
    memory = 0.0;
  }
  @Override
  public void add(double number){
    memory += number;
  }
  @Override
  public void multiply(double number){
    memory *= number;
  }
}
```

EXCEPTIONS

- Use the throw statement to throw exceptions:

  ```
  throw <theException>;
  ```

- Any instance of a class subclassing `Throwable` can be thrown.
  - Typically extend the class `Exception` for your own exceptions.
- Methods throwing exceptions:

  ```
  public void methodName() throws TheException{ }
  ```

  - `RunnableException`s don't need this.
- Handle thrown exceptions with the `try`, `catch` and `finally` statements.

EXCEPTIONS EXAMPLE

```
class MyException extends Exception{
  public MyException(String msg){
    super(msg);
  }
}
```

```
try{
  mightThrowException();
}catch(MyException e){
  // Handle the exception.
  System.out.println(e.getMessage());
}catch(ExceptionName e){
  e.printStackTrace();
}finally{
  // Clean up!
}
```

GENERIC CLASSES

- A generic class has type parameters:

```
class ClassName<T>{ ... }
```

- The one using the class specifies which type to use:

```
ClassName<AClass> o = new ClassName<>()
```

- Typically used for collections:

```
ArrayList<MyClass> list = new ArrayList<>();
list.add(new MyClass());
list.add(new MyClass());
```

LIST EXAMPLE

```
public class Human{
  public int age;
  public Human(int age){
    this.age = age;
  }
}
```

```
public class TestClass{
  public static void main(String[] args){
    ArrayList<Human> humans = new ArrayList<>();
    humans.add(new Human(23));
    humans.add(new Human(24));
    Human human = humans.get(1);
    humans.remove(0);
    humans.remove(human);
  }
}
```

EXAMPLE

```
public class Pair<T1, T2>{
  private T1 field1;
  private T2 field2;
  public Pair(T1 field1, T2 field2){
    this.field1 = field1;
    this.field2 = field2;
  }
  public T1 get1(){ return field1; }
  public T2 get2(){ return field2; }
}
```

```
public class TestClass{
 public void Test(){
  Pair<ClassA, ClassB> p = new Pair<>(
   new ClassA(),
   new ClassB()
  );
  ClassA a = p.get1();
  ClassB b = p.get2();
 }
}
```

GENERIC CLASSES

- Primitive datatypes can't be used as generic type parameters.
- But each primitive datatype have a corresponding class:
  - `int` has the class `Integer`.
  - `double` has the class `Double`.
  - …

EXAMPLE

```
public class TestClass{
  public void Test(){
    Pair<Integer, Double> p = new Pair<>(
      new Integer(20),
      new Double(2.32)
    );
    int a = p.get1().intValue();
    double b = p.get2().doubleValue();
  }
}
```

GENERIC CLASSES

Java 5.0 added:

- Autoboxing:
  - Convert a primitive type to corresponding class automatically.
- Unboxing:
  - Convert a class to the corresponding primitive type automatically.

EXAMPLE

```
public class TestClass{
  public void Test(){
    Pair<Integer, Double> p = new Pair<>(
      20,
      2.32
    );
    int a = p.get1();
    double b = p.get2();
  }
}
```

DOWNCASTING

```java
public abstract class ClassA{}
```

```java
public class ClassB1 extends ClassA{
  public int b1 = 1;
}
```

```java
public class ClassB2 extends ClassA{
  public int b2 = 23;
}
```

```java
public class Test{
  public int sum(List<ClassA> things){
    int sum = 0;
    for(ClassA thing : things){
      if(thing instanceof ClassB1){
         ClassB1 t = (ClassB1) thing;
         sum += t.b1;
      }else{
         ClassB2 t = (ClassB2) thing;
         sum += t.b2
      }
    }
    return sum;
  }
}
```

DOWNCASTING

Avoid it if possible.

- E.g. by using polymorphism.

```
public class Test{
  public int sum(List<ClassA> things){
    int sum = 0;
    for(ClassA thing : things){
      sum += thing.getInt();
    }
    return sum;
  }
}
```

```
public abstract class ClassA{
  public abstract int getInt();
}
```

```
public class ClassB1 extends ClassA{
  public int b1 = 1;
  @Override
  public int getInt(){ return b1; }
}
```

```
public class ClassB2 extends ClassA{
  public int b2 = 23;
  @Override
  public int getInt(){ return b2; }
}
```

STRINGS

- Handled through the class `String` from the package `java.lang`.
  - Can be created with double quotes:
    `"This is a string!"`
- Are immutable.
- Concatenate strings with the + operator.
  - `"This is "+"a string!"` → `"This is a string!"`
  - Any value can be used with the concatenation operator.
    - The `toString()` method is called on the object.
- The == operator compares references, use the `equals` method instead.

THE STATIC KEYWORD

```
public class Counter{
  private static int value = 0;
  public static void inc(int amount){
    value += amount;
  }
  public static int getValue(){
    return value;
  }
}
```

```
public class TestProgram{
  public static void main(String[] args){
    Counter.inc(12);
    System.out.println(Counter.getValue());
  }
}
```

THE STATIC KEYWORD

```java
public class MyClass{
  private int age = 23;
  public class MyInnerClass2{
    // I can access instance variables from MyClass!
  }
  public static class MyInnerClass{
    // I can't do that ☹
  }
}
```

THE STATIC KEYWORD

```
public class MyClass{
  public static ArrayList<Integer> ints = new ArrayList<>();
  static{
    ints.add(1);
    ints.add(2);
    ints.add(3);
  }
}
```

ENUMS

```java
public class MyClass{
  public enum Day{ MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
                   FRIDAY, SATURDAY, SUNDAY }
  public static void main(String[] args){
    Day today = Day.SATURDAY;
    if(today == Day.MONDAY){
      System.out.println("Time to work!");
    }
  }
}
```