Lecture 3

Java SE – Programming

presentation

Java Programming – Software App Development

Assoc. Prof. Cristian Toma Ph.D.

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

Cristian Toma – Business Card

Cristian Toma

IT&C Security Master

Dorobantilor Ave., No. 15-17
010572 Bucharest - Romania
http://ism.ase.ro
cristian.toma@ie.ase.ro
T +40 21 319 19 00 - 310
F +40 21 319 19 00

Agenda for Lecture 3 – Summary of OOP in JSE

Java – Inheritance, Overriding versus Overloading, Dynamic Binding + Robocode

Java OOP Inheritance & Polymorphism + Robocode

1.1 Java Objects and Classes

Object Oriented Programming (OOP):
involves software development using objects.

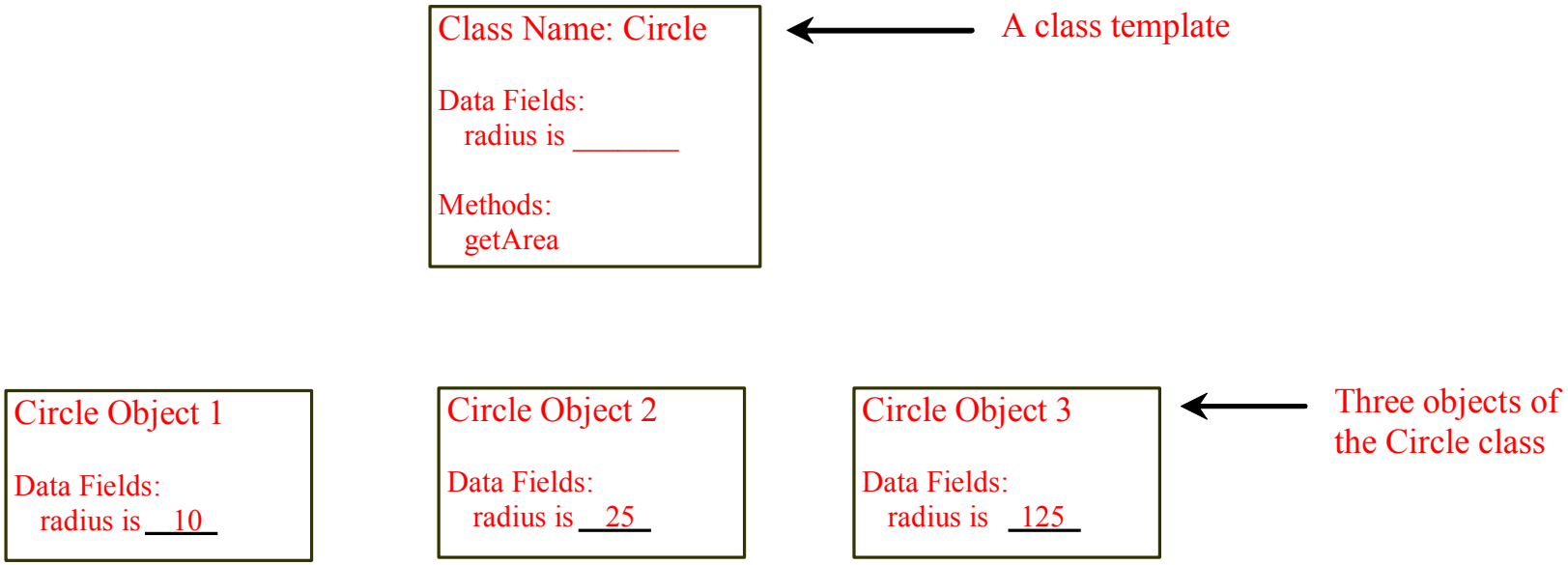
An ***object*** represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique:

- identity,
- state, and
- behaviors.

The ***state*** of an object consists of a set of ***data fields*** (also known as *properties*) with their current values. The ***behavior*** of an object is defined by a set of methods.

1.1 Java Objects and Classes

Objects: Instances of a class.



An **object** has both a *state* and *behavior*. The *state* defines the object, and the *behavior* defines what the object does.

1.1 Java Objects and Classes

Classes are constructs that define objects of the same type. A **Java class** uses *variables* to define *data fields* and *methods* to define *behaviors*. Additionally, a class provides a *special type of methods*, known as *constructors*, which are invoked to construct objects from the class.

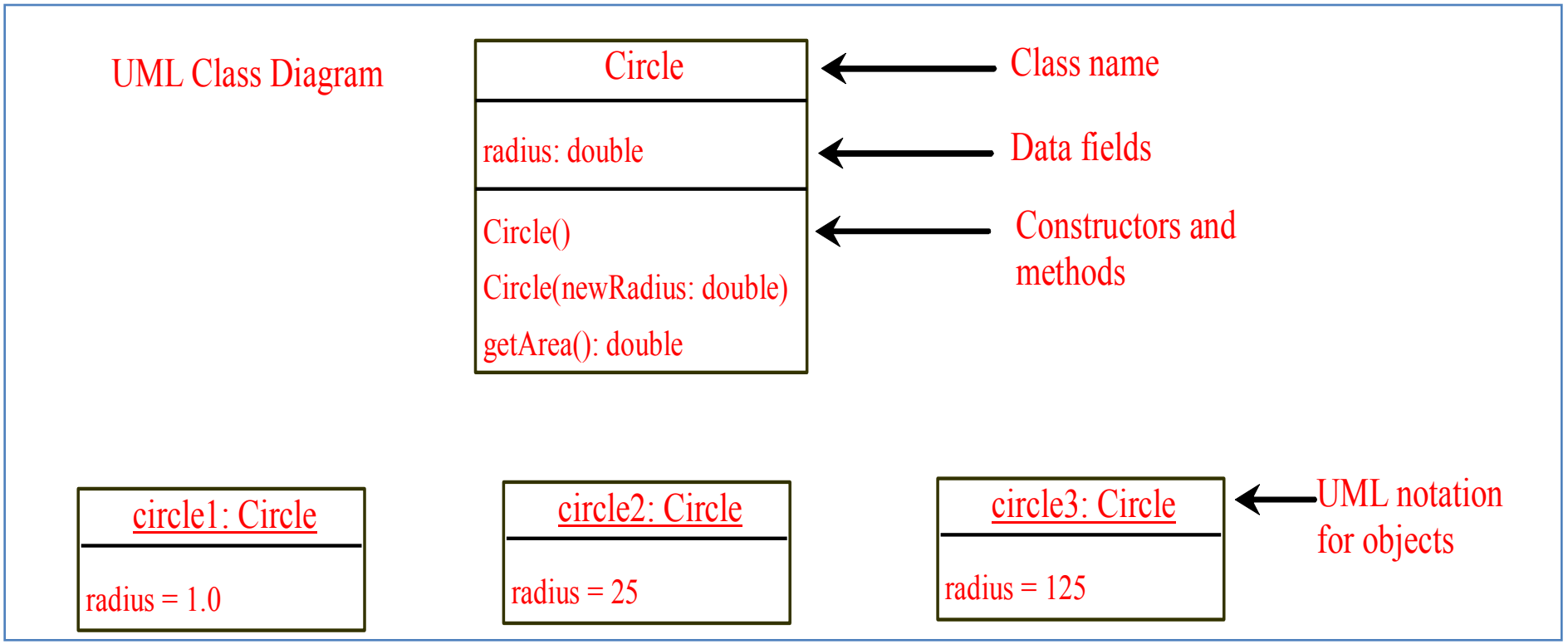
```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

Data field

Constructors

Method

UML: Unified Modeling Language – Class Diagram



Constructors:

A constructor with no parameters is referred to as a *no-arguments constructor*.

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

A class may be declared without constructors. In this case, a no-arguments constructor with an empty body is implicitly declared in the class. This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly declared in the class*.

```
Circle() {
}
Circle(double newRadius) {
  radius = newRadius;
}
```

Declaring / Creating Objects in a single step.

```
ClassName objectRefVar = new ClassName();
```

Assign object reference — Create an object

Example:

```
Circle myCircle = new Circle();
```

Accessing Objects

- Referencing the object's data:

  `objectRefVar.data`

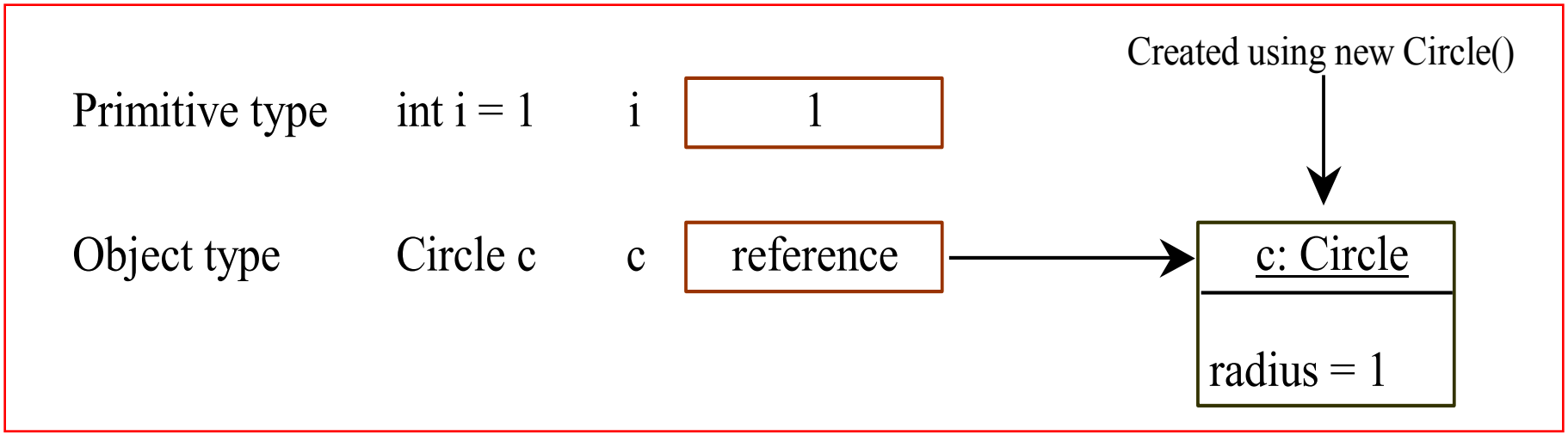
  e.g., `myCircle.radius`

- Invoking the object's method:

  `objectRefVar.methodName(arguments)`

  e.g., `myCircle.getArea()`

Differences between Variables of Primitive Data Types and Object Types

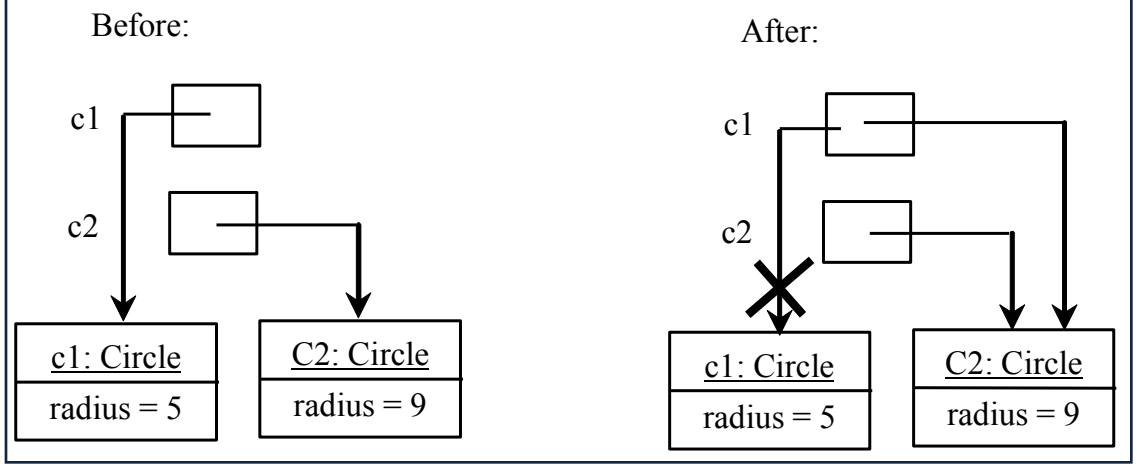


Copying Variables of Primitive Data Types and Object Types

Primitive type assignment i = j

As shown in the figure below, after the assignment statement c1 = c2, c1 points to the same object referenced by c2. The object previously referenced by c1 is no longer referenced. This object is known as garbage. Garbage is automatically collected by JVM.

Object type assignment c1 = c2 (shallow copy)



Instance (Objects), Variables and Methods

Instance variables (**relation "has a"** at data field level) belong to a specific instance.

Instance methods are invoked by an instance of the class.

Static Variables, Constants and Methods

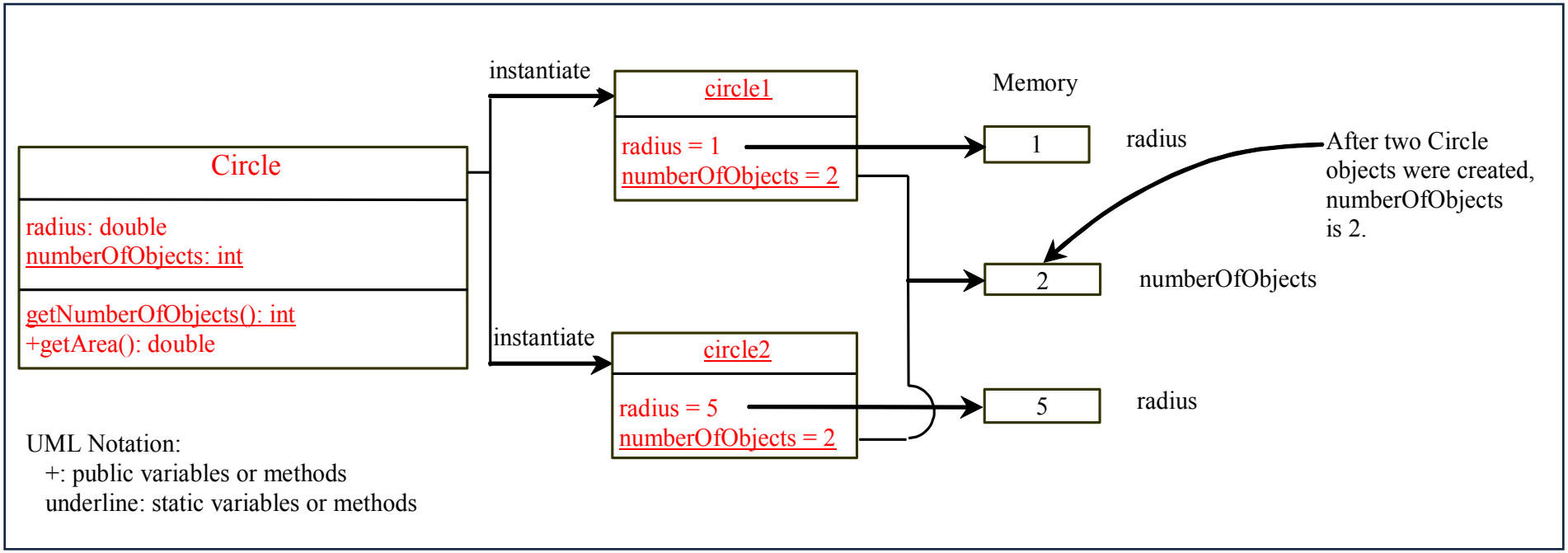
Static variables are shared by all the instances of the class.

Static methods are not tied to a specific object.

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

To declare static variables, constants, and methods, use the *static* modifier.

Static: Variables, Constants and Methods



Visibility Modifiers and Accessor/Mutator Methods

By default, the class, variable, or method can be accessed by any class in the same package.

- **`public`**

  The class, data, or method is visible to any class in any package.

- **`private`**

  The data or methods can be accessed only by the declaring class.

The get and set methods are used to read and modify private properties.

1.1 Java Objects and Classes

Visibility Modifiers: The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the public modifier enables unrestricted access. Modifier protected preserves the access at inheritance chain level.

package p1;

```
public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

package p2;

```
public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

package p1;

```
class C1 {
  ...
}
```

```
public class C2 {
  can access C1
}
```

package p2;

```
public class C3 {
  cannot access C1;
  can access C2;
}
```

1.1 Java Objects and Classes

Note: An object cannot access its private members, as shown in (b). It is OK, however, if the object is declared in its own class, as shown in (a).

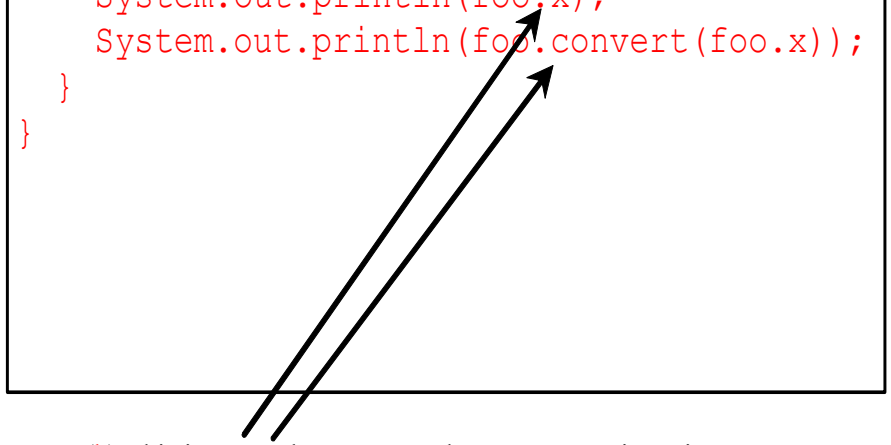
Why Data Fields Should Be private?

To protect data. + To make class easy to maintain.

```
public class Foo {
  private boolean x;

  public static void main(String[] args) {
    Foo foo = new Foo();
    System.out.println(foo.x);
    System.out.println(foo.convert());
  }

  private int convert(boolean b) {
    return x ? 1 : -1;
  }
}
```

(a) This is OK because object foo is used inside the Foo class

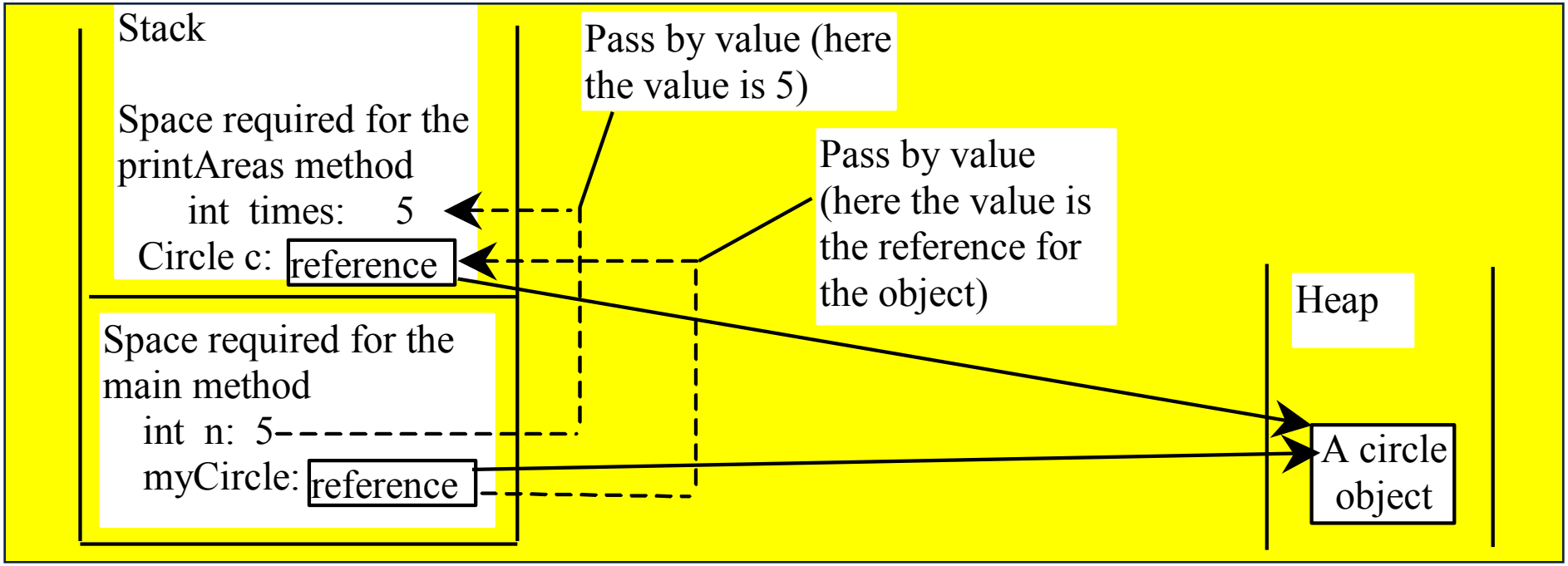
```
public class Test {
  public static void main(String[] args) {
    Foo foo = new Foo();
    System.out.println(foo.x);
    System.out.println(foo.convert(foo.x));
  }
}
```



(b) This is wrong because x and convert are private in Foo.

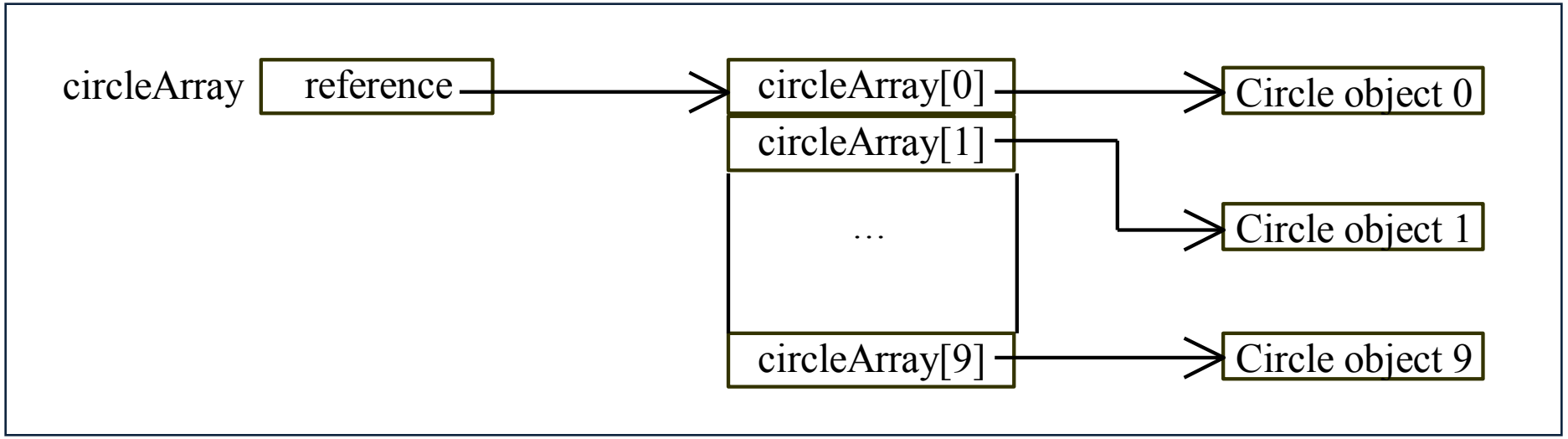
Passing Arrays / Objects and primitive data type variables to the methods

- ***Passing by value*** *for primitive type* value (the value is passed to the parameter)
- ***Passing by value*** *for reference type* value (the value is the reference to the object)



Array of the Objects: Circle class sample

```
Circle[] circleArray = new Circle[10];
```



Immutable Classes and Objects.

If the contents of an object cannot be changed once the object is created, the object is called an *immutable object* and its class is called an *immutable class*. If you delete the set method in the **Circle** class, the class would be immutable because radius is private and cannot be changed without a set method.

A class with all private data fields and without mutators is not necessarily immutable. For example, the following class **Student** has all private data fields and no mutators, but it is **mutable**.

What Class is immutable?

For a class to be immutable, it must mark all data fields private and provide no mutator methods and no accessor methods that would return a reference to a mutable data field object.

1.1 Java Objects and Classes

Immutable Classes and Objects.

```
public class Student {
  private int id;
  private BirthDate birthDate;

  public Student(int ssn,
      int year, int month, int day) {
    id = ssn;
    birthDate = new BirthDate(year, month, day);
  }

  public int getId() {
    return id;
  }

  public BirthDate getBirthDate() {
    return birthDate;
  }
}
```

```
public class BirthDate {
  private int year;
  private int month;
  private int day;

  public BirthDate(int newYear,
      int newMonth, int newDay) {
    year = newYear;
    month = newMonth;
    day = newDay;
  }

  public void setYear(int newYear) {
    year = newYear;
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, 1970, 5, 3);
    BirthDate date = student.getBirthDate();
    date.setYear(2010); // Now the student birth year is changed!
  }
}
```

1.1 Java Objects and Classes

This keyword plus the variables / objects *visibility scope*.

- The scope of instance and static variables is the entire class. They can be declared anywhere inside a class.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be initialized explicitly before it can be used.

The **this** keyword is the name of a reference that refers to an object itself. One common use of the **this** keyword is reference a class's *hidden data fields*.

Another common use of the **this** keyword to enable a constructor to invoke another constructor of the same class.

```
public class Foo {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    Foo.k = k;
  }
}
```

```
Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

Calling overloaded constructor

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

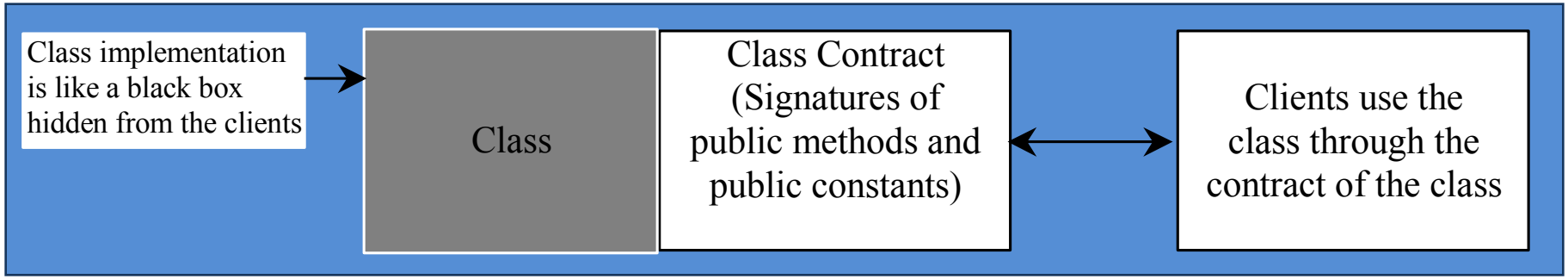
this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

Class *Abstraction* and *Encapsulation*

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



Designing a *Class*

Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1)
-loanAmount: double	The loan amount (default: 1000).
-loanDate: Date	The date this loan was created.
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate to this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years to this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount to this loan.
+getMonthlyPayment(): double	Returns the monthly payment of this loan.
+getTotalPayment(): double	Returns the total payment of this loan.

Designing a *Class*

- (Coherence) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.
- (Separating responsibilities) A single entity with too many responsibilities can be broken into several classes to separate responsibilities. The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities.
  - The **String** class deals with immutable strings;
  - The **StringBuilder** class is for creating mutable strings, and;
  - The **StringBuffer** class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.
- (Abstraction, Encapsulation and Inheritance – "has a" vs. "is a" relationship) Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it, design the properties to ensure that the user can set properties in any order, with any combination of values, and design methods to function independently of their order of occurrence.
- (Override methods) Provide a public no-arg constructor and *override* the equals method and the toString method defined in the **Object** class whenever possible.

Designing a *Class*

- Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. Always place the data declaration before the constructor, and place constructors before methods. Always provide a constructor and initialize variables to avoid programming errors.
- Using visibility modifiers:
  - Each class can present two contracts – one for the users of the class and one for the extenders of the class. Make the fields private and accessor methods public if they are intended for the users of the class. Make the fields or method protected if they are intended for extenders of the class. The contract for the extenders encompasses the contract for the users. The extended class may increase the visibility of an instance method from protected to public, or change its implementation, but you should never change the implementation in a way that violates that contract.
  - A class should use the private modifier to hide its data from direct access by clients. You can use get methods and set methods to provide users with access to the private data, but only to private data you want the user to see or to modify. A class should also hide methods not intended for client use.
- Use static modifiers: A property that is shared by all the instances of the class should be declared as a static property.

1. Java Inheritance and Polymorphism

Constructor Chaining Sample: Constructing an instance of a class invokes all the super-classes' constructors along the inheritance chain. This is called *constructor chaining*. Inheritance ensures **relation "is a"** in OOP.

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Java Inheritance and Polymorphism – Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

1. Java Inheritance and Polymorphism – Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

1. Java Inheritance and Polymorphism – Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee's no-arg constructor

1. Java Inheritance and Polymorphism – Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

1. Java Inheritance and Polymorphism – Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

1. Java Inheritance and Polymorphism – Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

1. Java Inheritance and Polymorphism – Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

1. Java Inheritance and Polymorphism – Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

1. Java Inheritance and Polymorphism – Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

1. Java Inheritance and Polymorphism

Are superclass's Constructor Inherited?

No. They are not inherited.

They are invoked explicitly or implicitly.

Explicitly using the **super** keyword.

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.* A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

CAUTION

You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Declaring a Subclass

A subclass extends properties and methods from the superclass (re-use code). You can also:

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

Overriding Methods in the Superclass

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

```
public class SpecialStudent extends Student {
  // Other methods are omitted


  /** Override the toString method defined in Student
*/
  public String toString() {
    return super.toString() + "\n is Erasmus student "
+ erasmus;
  }

}
```

NOTE

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

The Object Class and Its Methods

Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {
  ...
}
```

Equivalent

```
public class Circle extends Object {
  ...
}
```

The toString() method in Object

The toString() method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like Loan@15037e5 . This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.

1. Java Inheritance and Polymorphism

```
public class PolymorphismDemo {
public static void main(String[] args)
{
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
}

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student
{
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Polymorphism, Dynamic Binding and Generic Programming

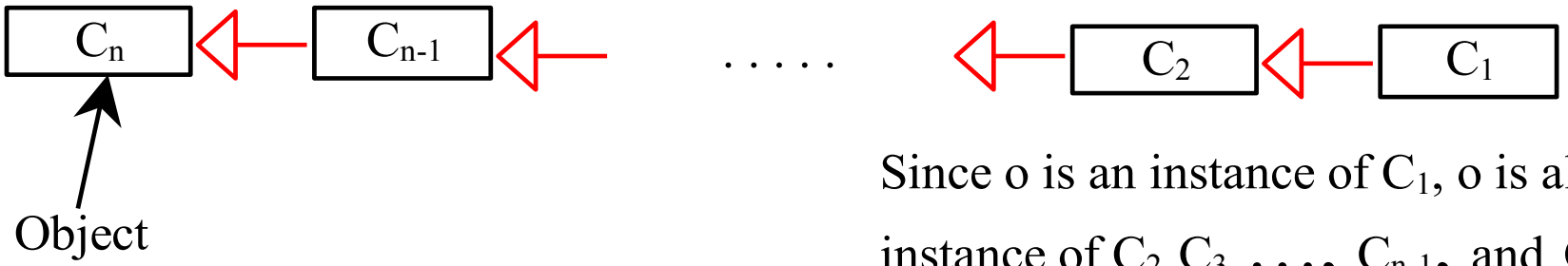
Method m takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime.

1. Java Inheritance and Polymorphism

```
public class PolymorphismDemo {
  public static void main(String[] args)
{
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Generic Programming

Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String). When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically. Starting with JDK 5, GENERICS programming has "special" syntax.

Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

Why Casting Is Necessary?

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Student b = o;
```

A compilation error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't? This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not so clever to know it. To tell the compiler that o is a Student object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

The `instanceof` Operator

Use the `instanceof` operator to test whether an object is an instance of a class (we will see it in reflection):

```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of
  Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ...
}
```

TIP

To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange. An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

The `equals` Method

The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

For example, the equals method is overridden in the Circle class.

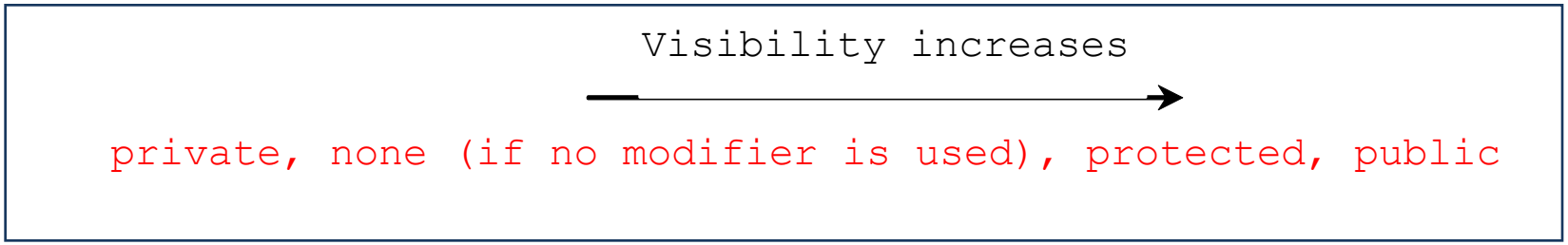
```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

NOTE

The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The equals method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.

The `protected` Modifier

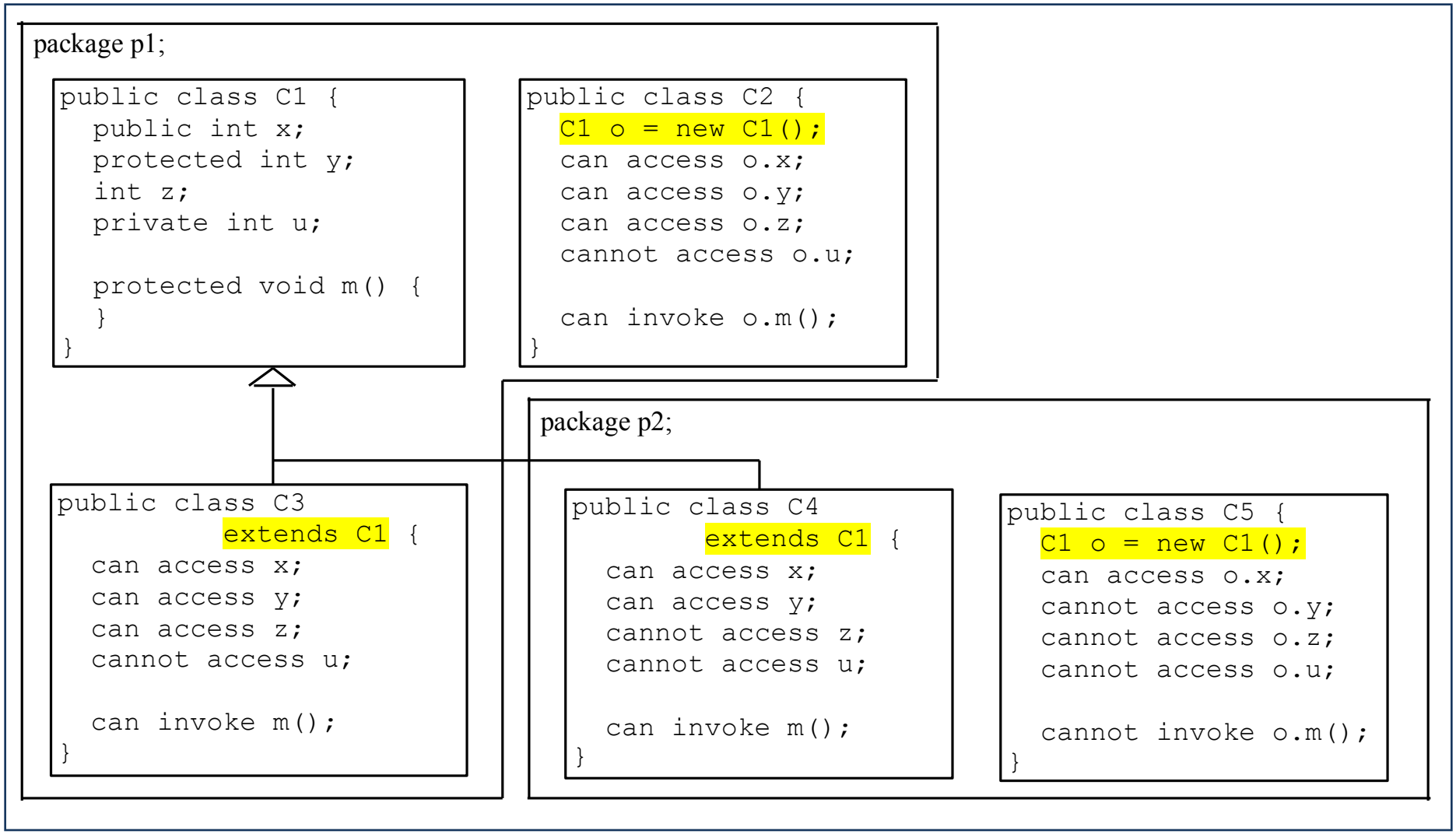
- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public



Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers



A Subclass Cannot Weaken the Accessibility

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

NOTE - The `final` Modifier

The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

- The `final` class cannot be extended:

```
final class Math {
  ...
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

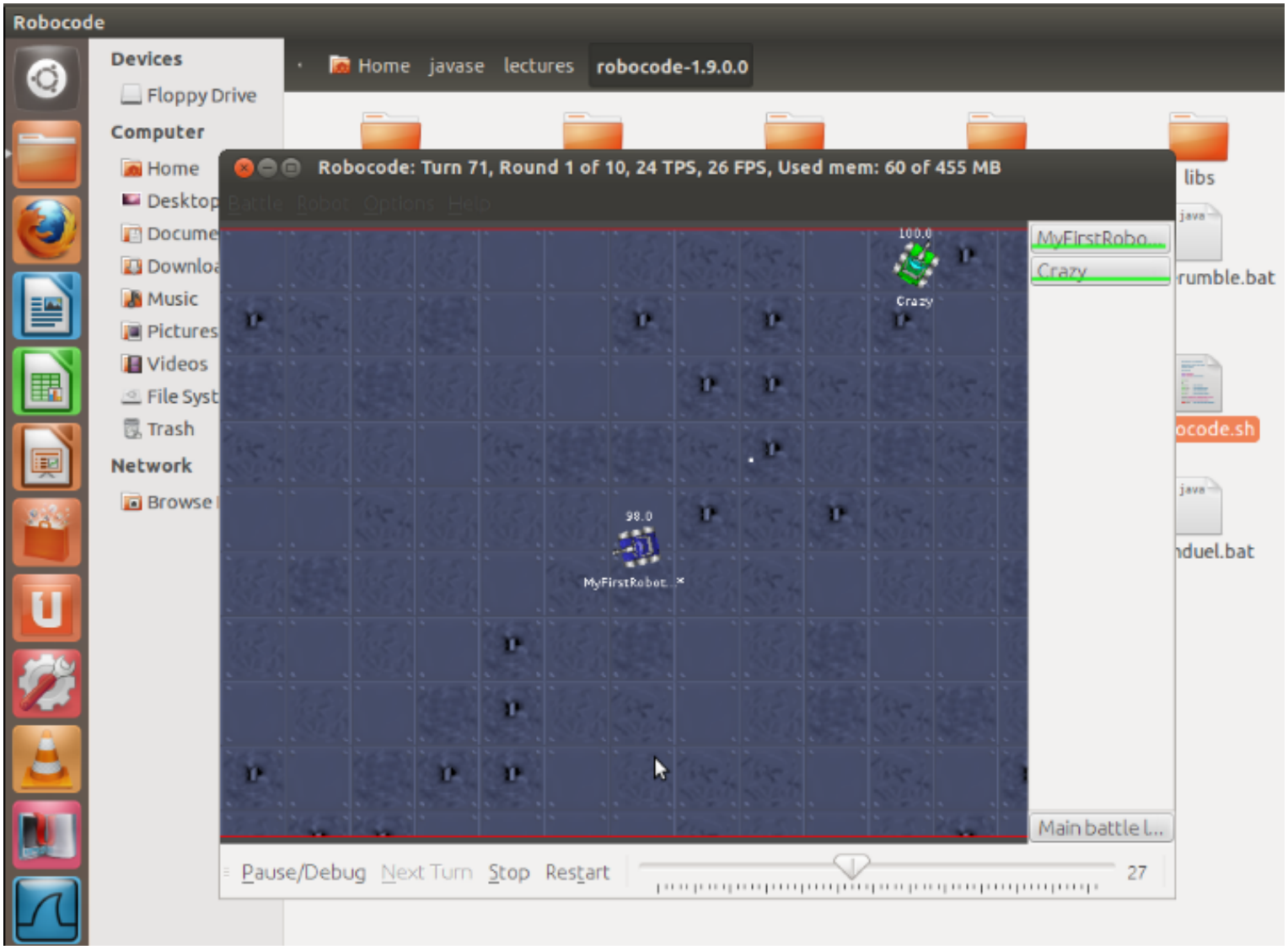
1. Java Robocode

Anatomy of a Robot + Demo + Assignments @ acs.ase.ro

```
package yourname;
import robocode.*;

/**
 * Description of Robot
 */
public class MyRobot extends Robot
{
   /**
    * Description of run method
    */
   public void run() {
       //the robots behaviour
   }
}
```

1. Java Robocode

Anatomy of a Robot + Demo + Assignments 01-05 @ acs.ase.ro



Section Conclusion

Fact: Inheritance in Java

In few **samples** it is simple to remember: Object Oriented Programming in Java – class, abstraction and encapsulation, object, object's deep vs. shallow copy, interface, interface as type, "has a" versus "is a" relationship, abstract class, inheritance, polymorphism + Robocode for fun learning Java OOP.

Java Strings classes – String, StringBuffer, StringBuilder

Java Strings & Command Line Arguments

2.1 String classes

The `String` Class

- Constructing a String:
  - `String message = "Welcome to Java";`
  - `String message = new String("Welcome to Java");`
  - `String s = new String();`
- Obtaining String length and Retrieving Individual Characters in a string
- String Concatenation (***concat***)
- Substrings (***substring(index)***, ***substring(start, end)***)
- Comparisons (***equals***, ***compareTo***)
- String Conversions
- Finding a Character or a Substring in a String
- Conversions between Strings and Arrays
- Converting Characters and Numeric Values to Strings

Constructing Strings

String newString = new String(stringLiteral);

String message = new String("Welcome to Java");

Since strings are used frequently, Java provides a shorthand initializer for creating a string:

String message = "Welcome to Java";

Strings Are Immutable

A String object is immutable; its contents cannot be changed.

Does the following code change the contents of the string?

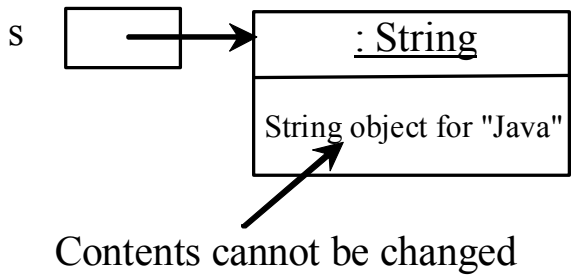
```
String s = "Java";
s = "HTML";
```

2.1 String classes

```
String s = "Java";
s = "HTML";
```

Trace Code

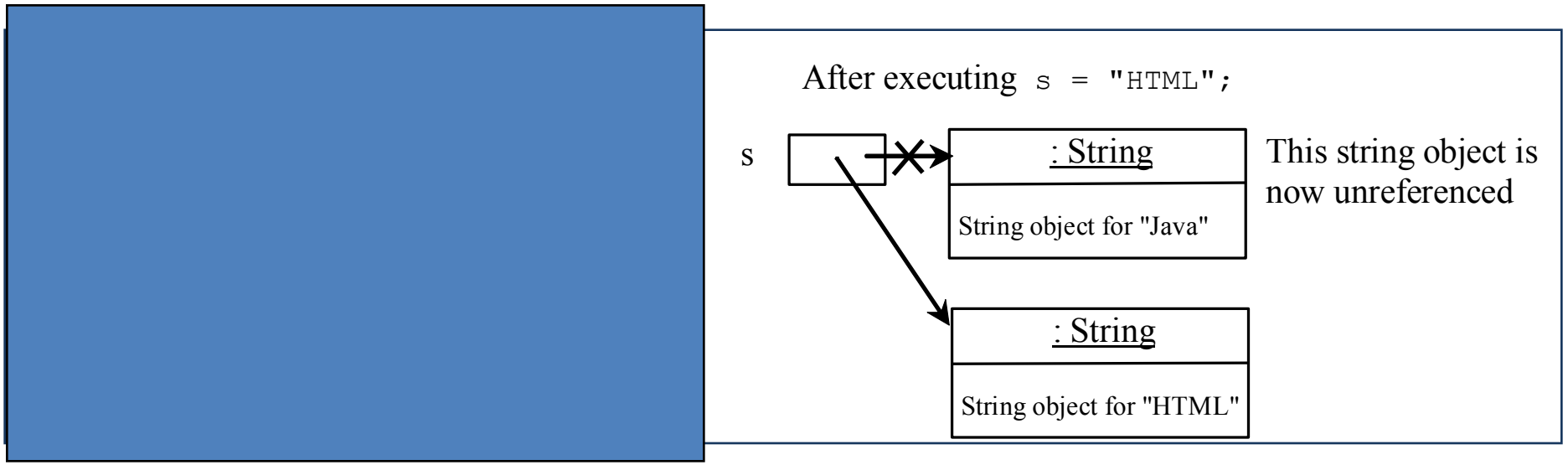
After executing `String s = "Java";`



2.1 String classes

```
String s = "Java";
s = "HTML";
```

Trace Code

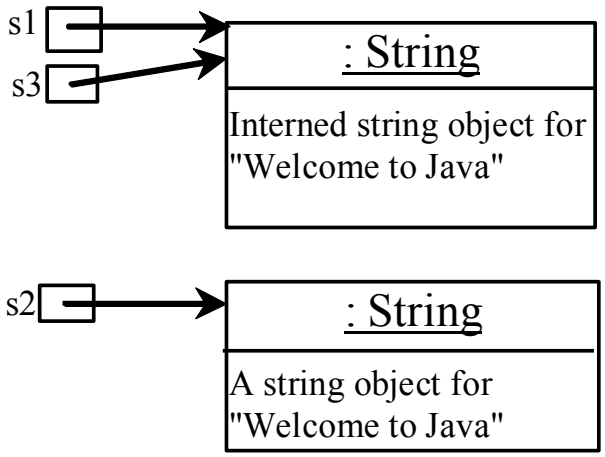


2.1 String classes

Interned Strings

Since strings are immutable and are frequently used, to improve efficiency and save memory, the JVM uses a unique instance for string literals with the same character sequence. Such an instance is called *interned*. For example, the following statements:

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```



display

s1 == s is false

s1 == s3 is true

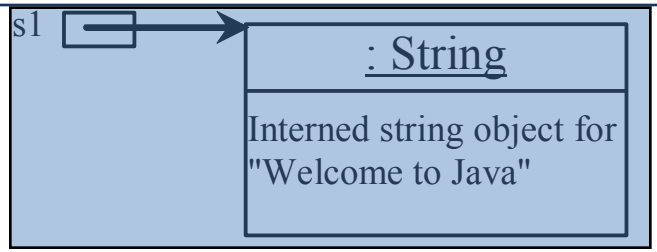
A new object is created if you use the new operator.

If you use the string initializer, no new object is created if the interned object is already created.

2.1 String classes

Trace Code

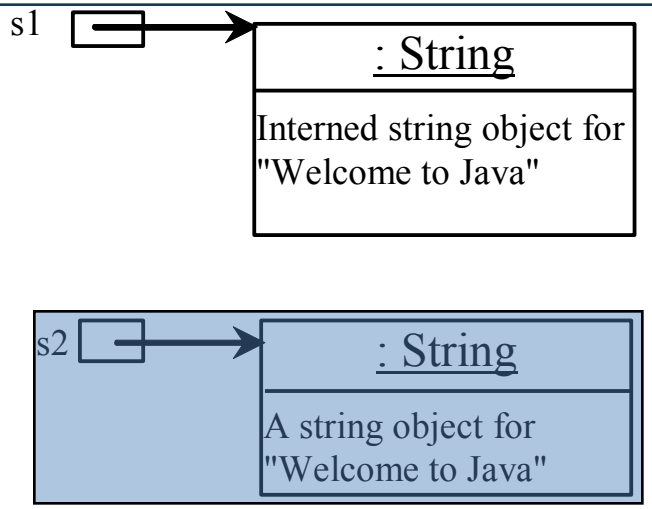
```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";
```



2.1 String classes

Trace Code

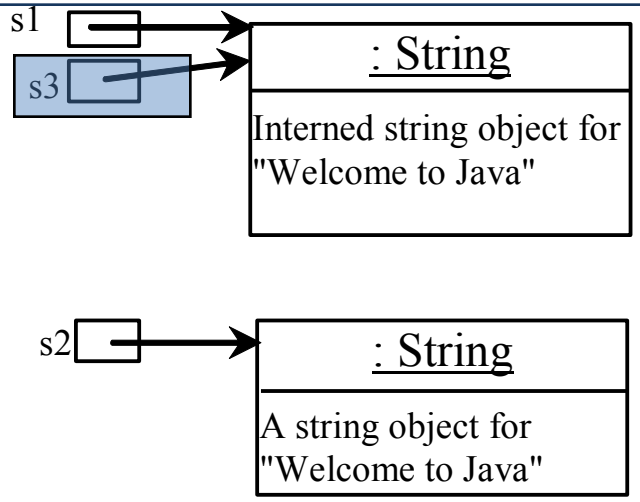
```
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
```



2.1 String classes

Trace Code

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";
```



2.1 String classes

String Comparisons

java.lang.String	
+equals(s1: String): boolean	Returns true if this string is equal to string s1.
+equalsIgnoreCase(s1: String): boolean	Returns true if this string is equal to string s1 case-insensitive.
+compareTo(s1: String): int	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1.
+compareToIgnoreCase(s1: String): int	Same as compareTo except that the comparison is case-insensitive.
+regionMatches(toffset: int, s1: String, offset: int, len: int): boolean	Returns true if the specified subregion of this string exactly matches the specified subregion in string s1.
+regionMatches(ignoreCase: boolean, toffset: int, s1: String, offset: int, len: int): boolean	Same as the preceding method except that you can specify whether the match is case-sensitive.
+startsWith(prefix: String): boolean	Returns true if this string starts with the specified prefix.
+endsWith(suffix: String): boolean	Returns true if this string ends with the specified suffix.

String Comparisons

- **equals**

```
String s1 = new String("Welcome");
String s2 = "welcome";

if (s1.equals(s2)){
  // s1 and s2 have the same contents
}

if (s1 == s2) {
  // s1 and s2 have the same reference
 }
```

String Comparisons, cont.

- **compareTo**(Object object)

```
String s1 = new String("Welcome");
String s2 = "welcome";

if (s1.compareTo(s2) > 0) {
  // s1 is greater than s2
}
else if (s1.compareTo(s2) == 0) {
  // s1 and s2 have the same contents
}
else
   // s1 is less than s2
```

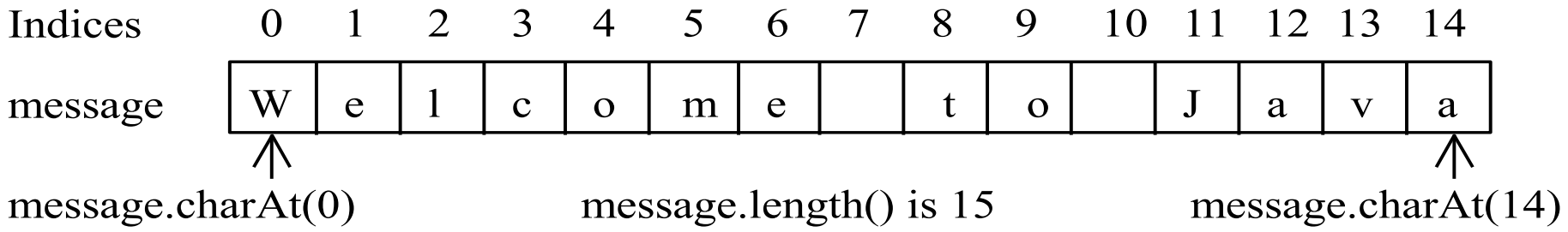
2.1 String classes

String Length, Characters, and Combining Strings

java.lang.String	
+length(): int	Returns the number of characters in this string.
+charAt(index: int): char	Returns the character at the specified index from this string.
+concat(s1: String): String	Returns a new string that concatenate this string with string s1.

Retrieving Individual Characters in a String

- Do not use `message[0]`
- Use `message.charAt(index)`
- Index starts from `0`



String Concatenation

```
String s3 = s1.concat(s2);
```

String s3 = s1 + s2;

s1 + s2 + s3 + s4 + s5 same as
(((s1.concat(s2)).concat(s3)).concat(s4)).concat(s5);

Extracting Substrings

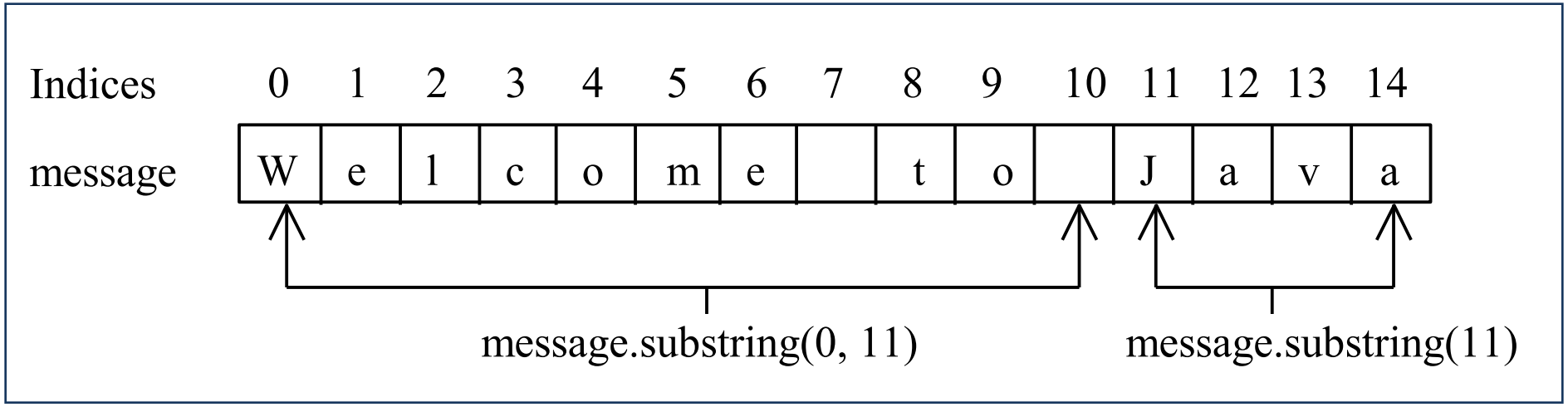
java.lang.String	
+subString(beginIndex: int): String	Returns this string's substring that begins with the character at the specified beginIndex and extends to the end of the string, as shown in Figure 8.6.
+subString(beginIndex: int, endIndex: int): String	Returns this string's substring that begins at the specified beginIndex and extends to the character at index endIndex – 1, as shown in Figure 8.6. Note that the character at endIndex is not part of the substring.

2.1 String classes

Extracting Substrings

You can extract a single character from a string using the charAt method. You can also extract a substring from a string using the substring method in the String class.

```
String s1 = "Welcome to Java";
String s2 = s1.substring(0, 11) + "HTML";
```



2.1 String classes

Converting, Replacing, and Splitting Strings

java.lang.String	
+toLowerCase(): String	Returns a new string with all characters converted to lowercase.
+toUpperCase(): String	Returns a new string with all characters converted to uppercase.
+trim(): String	Returns a new string with blank characters trimmed on both sides.
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching character in this string with the new character.
+replaceFirst(oldString: String, newString: String): String	Returns a new string that replaces the first matching substring in this string with the new substring.
+replaceAll(oldString: String, newString: String): String	Returns a new string that replace all matching substrings in this string with the new substring.
+split(delimiter: String): String[]	Returns an array of strings consisting of the substrings split by the delimiter.

2.1 String classes

Examples

"Welcome".toLowerCase() returns a new string, welcome.

"Welcome".toUpperCase() returns a new string, WELCOME.

" Welcome ".trim() returns a new string, Welcome.

"Welcome".replace('e', 'A') returns a new string, WAlcomA.

"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.

"Welcome".replace("e", "AB") returns a new string, WABlcomAB.

"Welcome".replace("el", "AB") returns a new string, WABlcome.

Splitting a String

```
String[] tokens = "Java#HTML#Perl".split("#", 0);
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

Displays: Java HTML Perl

2.1 String classes

Matching, Replacing and Splitting by Patterns

You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature, commonly known as ***regular expression***. Regular expression is complex to beginning students. For this reason, two simple patterns are used in this section.

```
"Java".matches("Java");
"Java".equals("Java");

"Java is fun".matches("Java.*");
"Java is cool".matches("Java.*");
```

2.1 String classes

Matching, Replacing and Splitting by Patterns

The replaceAll, replaceFirst, and split methods can be used with a regular expression. For example, the following statement returns a new string that replaces $, +, or # in "a+b$#c" by the string NNN.

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

Here the regular expression [$+#] specifies a pattern that matches $, +, or #. So, the output is aNNNbNNNNNNc.

2.1 String classes

Matching, Replacing and Splitting by Patterns

The following statement splits the string into an array of strings delimited by some punctuation marks.

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

for (int i = 0; i < tokens.length; i++)
  System.out.println(tokens[i]);
```

2.1 String classes

Finding a Character or a Substring in a String

java.lang.String	
+indexOf(ch: char): int	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
+indexOf(ch: char, fromIndex: int): int	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
+indexOf(s: String): int	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
+indexOf(s: String, fromIndex: int): int	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
+lastIndexOf(ch: int): int	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
+lastIndexOf(ch: int, fromIndex: int): int	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
+lastIndexOf(s: String): int	Returns the index of the last occurrence of string s. Returns -1 if not matched.
+lastIndexOf(s: String, fromIndex: int): int	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.

2.1 String classes

Finding a Character or a Substring in a String

`"Welcome to Java".indexOf('W')` returns 0.

`"Welcome to Java".indexOf('x')` returns -1.

`"Welcome to Java".indexOf('o', 5)` returns 9.

`"Welcome to Java".indexOf("come")` returns 3.

`"Welcome to Java".indexOf("Java", 5)` returns 11.

`"Welcome to Java".indexOf("java", 5)` returns -1.

`"Welcome to Java".lastIndexOf('a')` returns 14.

2.1 String classes

Convert Character and Numbers to Strings

The String class provides several static valueOf methods for converting a character, an array of characters, and numeric values to strings. These methods have the same name valueOf with different argument types char, char[], double, long, int, and float. For example, to convert a double value to a string, use String.valueOf(5.44). The return value is string consists of characters ‘5’, ‘.’, ‘4’, and ‘4’.

2.1 String classes

The `Character` Class

java.lang.Character	
+Character(value: char)	Constructs a character object with char value
+charValue(): char	Returns the char value from this object
+compareTo(anotherCharacter: Character): int	Compares this character with another
+equals(anotherCharacter: Character): boolean	Returns true if this character equals to another
+isDigit(ch: char): boolean	Returns true if the specified character is a digit
+isLetter(ch: char): boolean	Returns true if the specified character is a letter
+isLetterOrDigit(ch: char): boolean	Returns true if the character is a letter or a digit
+isLowerCase(ch: char): boolean	Returns true if the character is a lowercase letter
+isUpperCase(ch: char): boolean	Returns true if the character is an uppercase letter
+toLowerCase(ch: char): char	Returns the lowercase of the specified character
+toUpperCase(ch: char): char	Returns the uppercase of the specified character

2.1 String classes

Examples

```
Character charObject = new Character('b');
```

charObject.compareTo(new Character('a')) returns 1
charObject.compareTo(new Character('b')) returns 0
charObject.compareTo(new Character('c')) returns -1
charObject.compareTo(new Character('d') returns –2
charObject.equals(new Character('b')) returns true
charObject.equals(new Character('d')) returns false

2.1 String classes

`StringBuilder` and `StringBuffer`

The `StringBuilder`/`StringBuffer` class is an alternative to the `String` class. In general, a StringBuilder/StringBuffer can be used wherever a string is used. StringBuilder/StringBuffer is more flexible than String. You can add, insert, or append new contents into a string buffer, whereas the value of a String object is fixed once the string is created.

`StringBuilder` Constructors

java.lang.StringBuilder	
+StringBuilder()	Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)	Constructs a string builder with the specified capacity.
+StringBuilder(s: String)	Constructs a string builder with the specified string.

2.1 String classes

Modifying Strings in the Builder

java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from startIndex to endIndex.
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array to the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: *aPrimitiveType*): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from startIndex to endIndex with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

2.1 String classes

Examples

stringBuilder.append("Java");

stringBuilder.insert(11, "HTML and ");

stringBuilder.delete(8, 11) changes the builder to Welcome Java.

stringBuilder.deleteCharAt(8) changes the builder to Welcome o Java.

stringBuilder.reverse() changes the builder to avaJ ot emocleW.

stringBuilder.replace(11, 15, "HTML")
changes the builder to Welcome to HTML.

stringBuilder.setCharAt(0, 'w') sets the builder to welcome to Java.

2.1 String classes

The toString, capacity, length, setLength, and charAt Methods

java.lang.StringBuilder	
+toString(): String	Returns a string object from the string builder.
+capacity(): int	Returns the capacity of this string builder.
+charAt(index: int): char	Returns the character at the specified index.
+length(): int	Returns the number of characters in this builder.
+setLength(newLength: int): void	Sets a new length in this builder.
+substring(startIndex: int): String	Returns a substring starting at startIndex.
+substring(startIndex: int, endIndex: int): String	Returns a substring from startIndex to endIndex-1.
+trimToSize(): void	Reduces the storage size used for the string builder.

2.2 Main method parameters

Main Method Is Just a Regular Method

You can call a regular method by passing actual parameters. Can you pass arguments to main? Of course, yes. For example, the main method in class B is invoked by a method in A, as shown below:

```
public class A {
  public static void main(String[] args) {
    String[] strings = {"New York",
      "Boston", "Atlanta"};
    B.main(strings);
  }
}
```

```
class B {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

2.2 Main method parameters

Command-Line Parameters

```
class TestMain {
  public static void main(String[] args) {
  ...
  }
}

java TestMain arg0 arg1 arg2 ... argn
```

Processing Command-Line Parameters

In the main method, get the arguments from `args[0]`, `args[1]`, ..., `args[n]`, which corresponds to `arg0`, `arg1`, ..., `argn` in the command line.

Section Conclusions

Java String class produces immutable objects that are thread-safe and interned

Can be used instead a more flexible approach with StringBuilder and StringBuffer classes

A lot of operation are implemented for strings manipulation

Command line arguments and main method – entry point

Java String Summary
for easy sharing

Java OOP Q&A – Share knowledge, Empowering Minds

Communicate & Exchange Ideas

3. Summary of Java SE - OOP

Analogy with C++ OOP has been reloaded in this lecture

What is a class?

What is a package of classes in Java?

What is an object / instance?

How many bytes has an object inside JVM?

Why do we need clone, equals and hash methods?

Demo & memory model for the ***Certificate*** Java class

3. Summary of Java SE - OOP

Polymorphism – "the ability to have multiple forms" is obtaining through:

- The overriding (not overloading; overloading is a form of polymorphism) of the methods from a Java class
- "Pure" form of polymorphism has to have the following conditions:
  - Inheritance mechanism – "***extends*** is the key word"
  - Virtual Methods – "in Java by default"
  - Overriding the virtual methods
  - Using the pointer / references to the objects – "in Java by default"

Interface – "contract between objects of the class that implements the interface and another objects of another classes that interact with objects from the implementation class" – has the following features:

- static fields
- Static and non-static methods prototypes – declaration but NOT implementation
- The implementation class use the "***implements***" keyword
- It is possible to have interface as type – declare objects with type interface but you must call the constructor methods from a real Java class

3. Summary of Java SE - OOP

Abstract Class – "a class that have at least one method abstract" – has the following features:

- At least one abstract method – keyword "***abstract***"
- May contain standard static and non-static fully implemented methods
- You may declare objects from an abstract class but you can NOT instantiate them, you should use constructor method from a real Java class

Pay attention @: Objects vs. vectors / arrays of objects + null pointer exception

3. Summary of Java SE - OOP

What are the advantages of ***derivation and inheritance*** in object-oriented programming in Java?

What are the advantages of ***polymorphism***?

What is an ***interface*** or an ***abstract class***?

What are the advantages and disadvantages for using "***Interface as type***"?

Demo & memory model for the ***Vehicle, Plane, Car*** in C++/Java classes

****** DEMO: C++ sample with Vehicle, Plane and Car and next lecture discuss more Java samples about: classes, interfaces, polymorphism …***

ASSIGNMENT 04 - Problem: Develop in Java, "Matrix" class and write a program that uses Matrix objects.

ASSIGNMENT 01 - 05: @ http://acs.ase.ro/java

Robocode Examples Discussion

Questions & Answers!

Thanks!

JSE – Java Standard Edition Programming
End of Lecture 3