326.041 (2015S) – Practical Software Technology

(Praktische Softwaretechnologie)

GC, Packages, Polymorphism

Alexander Baumgartner
Alexander.Baumgartner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Ada Lovelace

Figure: Ada Lovelace – The First Computer Programmer.

Memory Management

- Java manages the memory. There is **no explicit destructor** method.
- When using the keyword **new**, memory will be allocated for the object to be created.
- Unused objects are deleted by a process known as **garbage collection**
- JVM automatically runs GC periodically.
  - Identifies objects no longer in use (no references).
  - Finalizes those objects (deconstructs them).
  - Frees up memory used by destroyed objects.
  - Defragments memory.
- GC introduces overhead.
  - Avoid unnecessary object creation and deletion.

Memory Management


```java
public class Test {
    public static void main(String[] args) {
        Person p = new Person("Tina", 22, false);
        ...
        p = new Person("Max", 11, true);
        // No more reference to Tina ⇒ GC frees memory
        ...
    }
}
```

GC detects and frees unused objects.

- Reduces the size/complexity of the source code.
- Prevents deallocation of objects that are still in use.
- Prevents double-freeing objects.
- ...

Packages for Modular Programming

- Each java class is part of a **package**.
- Packages provide **modular programming** in Java.
- Similar to the modules of Modula.
- The root of a package structure is called **default package**.
- Every source file provides information about the package it belongs to. No declaration means: "I belong to the default package."

```
public class Test {
    ...
}
```

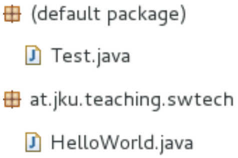
No package declaration ⇒ Class belongs to the default package.

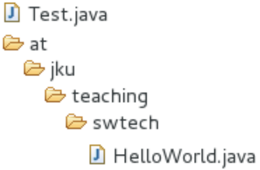
```
package at.jku.teaching.swtech; // FIRST STATEMENT
public class HelloWorld {
    ...
}
```

This class belongs to the package at.jku.teaching.swtech.

Packages and Folders

- **Package structures** correspond to folders in the file system, starting with an arbitrary root folder.






```
public class Test {
    ...
}
```

No package declaration ⇒ Class belongs to the default package.

```
package at.jku.teaching.swtech; // FIRST STATEMENT
public class HelloWorld {
    ...
}
```

This class belongs to the package at.jku.teaching.swtech.

Accessing Classes from other Packages

- We want to access HelloWorld from a class of another package:

```
package at.jku.teaching.swtech;
public class HelloWorld {
    ...
}
```

This class belongs to the package at.jku.teaching.swtech.

- The full name of a class consists of **package-name.class-name**:

```
    new at.jku.teaching.swtech.HelloWorld();
    at.jku.teaching.swtech.HelloWorld.STATIC_FIELD;
```

- The **import** statement can be used to shorten it:

```
package ... // Package declaration or default package
import at.jku.teaching.swtech.HelloWorld;
import java.util.*; // import package (bad practice)
...
    new HelloWorld();
    HelloWorld.STATIC_FIELD;
...
```

Important API Packages

- **java.lang:** Fundamental classes. No import needed.
  - **java.lang.reflect:** Dynamic invocation, Reflection.
- **java.util:** Array manipulation, Date and Time, Data Structures, Random numbers,...
  - **java.util.regex:** Regular expression.
  - **java.util.concurrent:** Concurrent programing.
- **java.io:** File operations.
- **java.nio:** New I/O framework for Java.
- **java.math:** Arbitrary precision arithmetics.
- **java.net:** Networking operations, sockets, DNS lookups.
- **java.security:** Encryption and decryption.
- **java.sql:** Java Database Connectivity.
- **java.awt:** Native GUI components.
- **javax.swing:** Platform-independent GUI components.
- ...

What is Polymorphism

- **Poly:** "many" from Greek $\pi o\lambda\acute{\upsilon}$ (poly)
- **Morp:** "form, figure, silhouette" from Greek $\mu o\rho\varphi\acute{\eta}$ (morphe)

Figure: Example for polymorphism in biology.

What is Polymorphism

- **Poly:** "many" from Greek πολύ (poly)
- **Morp:** "form, figure, silhouette" from Greek μορφή (morphe)

- Polymorphism by **overloading**:

```
// The method println from PrintStream is polymorph
System.out.println("Hello"); // Applicable to String
System.out.println(44);      // Applicable to int
System.out.println(true);    // Applicable to boolean
...                          // Applicable to ...
```

Figure: Example for polymorphism in Java I.

- Polymorphism is the ability to create a **function**, a variable, or an object that has more than one form.

Alexander.Baumgartner@risc.jku.at

What is Polymorphism

- **Poly:** "many" from Greek *πολύ* (poly)
- **Morp:** "form, figure, silhouette" from Greek *μορφή* (morphe)

- Polymorphism by **inheritance**:

```
Object o;            // Object o can hold any reference type
o = new Object();    // o can be of type Object
o = new String();    // o can be of type String
o = new int[]{};     // o can be of type int[]
...                  // o can be of type ...
```

Figure: Example for polymorphism in Java II.

- Polymorphism is the ability to create a function, a **variable**, or an object that has more than one form.

What is Polymorphism

- **Poly:** "many" from Greek *πολύ* (poly)
- **Morp:** "form, figure, silhouette" from Greek *μορφή* (morphe)

- Polymorphism by **generic types**:

```
// Generic programming = writing classes of variable type
List<String> l1 = new ArrayList<>(); // List of Strings
List<Person> l2 = new ArrayList<>(); // List of Persons
...                                  // List of ...
// Infinite number of different types with same behavior
```

Figure: Example for polymorphism in Java III.

- Polymorphism is the ability to create a function, a variable, or an **object** that has more than one form.

Why Polymorphism?

- Polymorphism encourages abstraction.
- More generalized programs can be extended more easily.
- E.g.: Online shopping application.
  - Multiple payment methods.
  - Might be implemented as separate classes because of differences.
  - Would require if-else statements everywhere to test for the different types of payment methods.
  - Solution: Define a base class PaymentMethod and then derive subclasses for each payment type.

Polymorphism by Overloading

Overloading: Compile time polymorphism (static binding).

- **Argument type:**

```
public static float abs(float val) {...
public static int abs(int val) {...
```

Operator overloading (e.g. "+") also belongs to this type.
In Java you can not define your own operators.

- **Argument count:**

```
public Person(String name, int age) {...
public Person(String name, int age, boolean man){..
```

Polymorphism by Inheritance

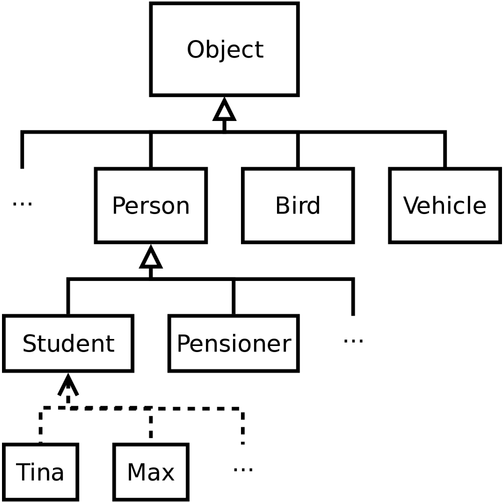


Figure: Inheritance is a tree of classes with the class Object as its root.

The Root java.lang.Object

- Has a default constructor.
- Offers some public methods:
  - boolean **equals**(Object obj): Tests for reference equality
  - String **toString**(): Returns "type-name@hash-code"
  - Class<?> **getClass**(): Returns the type information
  - int **hashCode**(): Returns a hash value (typically the internal address)
  - ...
- Offers some protected methods:
  - Object **clone**(): Returns a shallow copy of the object
  - ...

Polymorphism by Inheritance in Java

- Every class inherits from **java.lang.Object**.
- A class may inherit from another class by using the keyword **extends**.

```
public class Person {
    // Fields
    ...
    // Methods
    ...
}
public class Student extends Person {
    ...
}
```

- The class Student inherits from the class Person.
  - public fields and methods.
  - protected fields and methods.
- Inheritance **propagates** up the tree.
  - Student ⇒ Person ⇒ java.lang.Object.
  - Student inherits from java.lang.Object.

Overriding

- **Overriding:** Runtime polymorphism (dynamic binding).
- To override a method, the subclass method must have the **same signature**. E.g.: public String toString() {...}

```java
public class Person {
    ...
    public String toString() {
        return "I am a person and my name is " + name;
    }
}
public class Student extends Person {
    ...
    public String toString() {
        return "I am a student and my name is " + name;
    }
}
```

- Person overrides toString() from java.lang.Object.
- Student overrides toString() from Person.

Overriding

- **Overriding:** Runtime polymorphism (dynamic binding).
- To override a method, the subclass method must have the **same signature**. E.g.: public String toString() {...}

```
Object o;
o = new Object();
System.out.println(o); // type-name@hash-code
o = new Person(...);
System.out.println(o); // I am a person and...
o = new Student(...);
System.out.println(o); // I am a student and...
```

- Consult the **API source-code** to see how things work together!
- **println(o):** String.valueOf(o);
- **String.valueOf(o):** return (o == null) ? "null" : **o.toString()**;

Accessing Overridden Method/Field

- The keyword **super** allows to explicitly address the super class.
- The keyword **this** allows to explicitly address the actual class.

```java
public class Person {
    ...
    public String toString() {
        return "I am a person and my name is " + name;
    }
}
public class Student extends Person {
    ...
    public String toString() {
        return super.toString() + ". I study " + topic;
    }
}
```

Accessing Constructors

- The keyword **super** allows to explicitly address the super class.
- The keyword **this** allows to explicitly address the actual class.

```
public class Person {
    ...
    public Person(String name) { // default is woman
        this.name = name;
    }
    public Person(String name, boolean man) {
        this(name);            // FIRST STATEMENT!
        this.man = man;
    }
}
public class Student extends Person {
    ...
    public Student(String name,boolean man,String t) {
        super(name, man);   // FIRST STATEMENT!
        this.topic = t;     // this is optional
    }
}
```

Implicit Casting

- **Upcasting** of a **reference type** is using a more general type.
  - When an object reference is upcast, you can invoke only those methods declared by the more general type.
- On **primitive types** implicit casting is done for **widening** the type.

```
// Upcasting an object of type Student:
Student s = new Student(...);
Person p = s;   // Person is more general than Student
Object o = p;   // Object is more general than Person

// Widening a primitive integer value:
int i = 5;
double d = i;   // double is more general than int
```

Explicit Casting

- **Downcasting** of a **reference type** is using a more specific type.
- On **primitive types** explicit casting is necessary for **narrowing.**

```
// Downcasting an object of type Student:
Object o = new Student(...);
Person p = (Person)o;    // Person is more specific
Student s = (Student)p; // Student is more specific

// Narrowing a primitive double value:
double d = 5;
int i = (int)d;           // int is more specific
```

- Error will occur if o is not of type Student (ClassCastException).
- Data will be lost if d is a decimal number or out of the range of int.

Explicit Casting

- Sometimes losing data is desirable.
- Generate a random int $x \in \{0, 1, \ldots, 9\}$.
- Math.random() produces a double value $x \in [0, 1)$.

```
// int i = floor(x) for some random x ∈ [0,10)
int i = (int) (Math.random() * 10);
```

Abstract Classes

- **Abstract classes** are classes designed solely **for subclassing**.
  - They can not be instantiated.
  - They implement common sets of behavior, which are then shared by the concrete (instantiable) classes you derive from them.
  - You declare a class as abstract with the abstract modifier:

```
public abstract class Figure {
    protected double x, y...
```

- **Abstract methods** are methods with no body.
  - They declare the **method signature and return type**.
  - It is a **dummy method** for implementation of specific behavior.
  - Classes which contain abstract methods must also be abstract.
  - Instantiable subclasses provide implementations for abstract methods.
    - If a subclass does not provide implementations for all the abstract methods, then it must also be abstract (and it is not instantiable).
  - You declare a method as abstract with the abstract modifier and a semicolon terminator:

```
public abstract double getArea();
```

 Alexander.Baumgartner@risc.jku.at

Interfaces – Like Driving a Car

- Do you have a driving license?
- Do you want to make a new driving license for each car?
- No, because **there is a common interface**.
- You do not need to know how the engine works to drive a car.
- However, the car must be able to perform certain operations:
  - Go forward.
  - Slowdown/stop (break light).
  - Go in reverse.
  - Turn left (signal light).
  - Turn right (signal light).
  - ...

Interfaces

- Interfaces define method **signatures and return types**.
- They **do not provide any behavior**/implementation.
- Similar to abstract methods in abstract classes.
- An interface serves as a “contract” defining a set of capabilities through method signatures and return types.
- By implementing the interface, a class “advertises” that it provides the functionality required by the interface, and agrees to follow that contract for interaction.
- Interfaces are important for **Encapsulation and Modularity**.
  - Assume you need a sequence of unknown length (java.util.List).
  - It does not matter how it is implemented as long as it provides the necessary functionality.
  - You can exchange the implementation at any time.

Defining a Java Interface

- Use the **interface keyword** to define an interface in Java.
  - Naming convention is the same as for classes.
  - Interfaces contain definitions of abstract methods. E.g.:

```
public interface Shape {
    double getArea();
    double getPerimeter();
}
```

- Methods declared by an interface are **implicitly public abstract**.
  - You can omit either or both.
  - You must put a semicolon at the end.
- Interface might also declare and initialize **public static final fields**.
  - You can omit any or all of the public, static, and final keywords.

Implementing a Java Interface

- Use the **implements keyword** followed by the name. E.g.:

```
public class Circle implements Shape {...
    public double getArea() {...
```

- Abstract classes may (not) implement inherited abstract methods:

```
public abstract class Figure implements Shape {...
```

- A Java class can implement as many interfaces as needed:

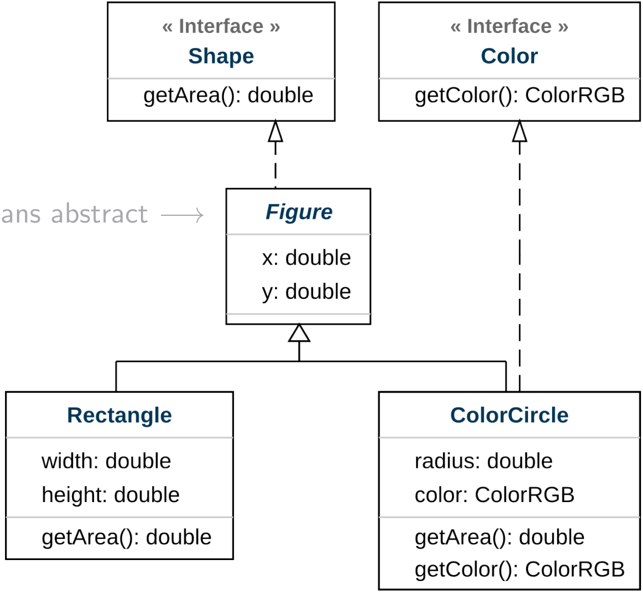
```
public class ColorCircle implements Shape, Color {...
    public double getArea() {...
    public ColorRGB getColor() {...
```

- A Java class can extend a base class and implement some interfaces:

```
public class ColorCircle extends Figure implements Color {
```

Example Class Diagram

Italic means abstract ⟶



Interface/Abstract Class & Polymorphism

- Every interface and every abstract class defines a type.
- Objects that implement an interface (extend an abstract class) can be assigned to reference variables typed to the interface (abstract class):

```
public class ColorCircle extends Figure implements Color {
```

```
ColorCircle cc = new ColorCircle(...);
Figure f = cc;
Color c = cc;
```

- Casting is the same as for "normal" classes.
- When an object reference is upcast, you can invoke only those methods declared by the interface / abstract class.

Abstract Classes vs. Interfaces

Abstract class	Interface
Fields that are not static and final	Fields are public, static, and final
Abstract and concrete methods	Only abstract methods
Any access modifier	Only public methods
You can extend only one class	Implement any number of interfaces

- Which should you use?
- Consider using an abstract class in the following situations:
  - Sharing code among several closely related classes.
  - Classes that extend your abstract class have common behavior or data.
- Consider using an interfaces in the following situations:
  - You expect that unrelated classes would implement your interface.
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  - You want to take advantage of multiple inheritance of type.

The **instanceof** Operator

- Remember **downcasting** of a **reference type** to a more specific type:

```
Object o;
...
Person p = (Person)o;    // Person is more specific
Student s = (Student)p; // Student is more specific
```

- Error will occur if o is not of type Student (ClassCastException).
- Use **instanceof** operator to test if an object is of a specific type:

```
if (o instanceof Person) {
    System.out.println("Object is of type Person");
}
if (o instanceof Student) {
    Student s = (Student)o;
    ...
```

- The instanceof operator works for classes, abstract classes, interfaces.

Object Equivalence

- java.lang.Object class provides an equals() method.
- Default behavior: Test for reference equality.
- E.g. two circles of same radius and same color at the same position.
- Override equals() to determine if two objects are equivalent.
- But be careful. The equality test must preserve the following properties:
  - **Symmetry**: a.equals(b) if and only if b.equals(a),
  - **Reflexivity**: a.equals(a),
  - **Transitivity**: if a.equals(b) and b.equals(c) then a.equals(c),
  - **Consistency with hashCode()**: Two equal objects must have the same hashCode() value.

Exercise

- Create a model and write a Java program to simulate an ecosystem.
  - March 19th: Release at the Moodle page.
  - March 26th: Discussion of your model and implementation strategy.
  - April 14th: Submission deadline.

See the guidance for this exercise on the Moodle page.