Programming languages Java

Polymorphism

Kozsik Tamás

ELTE
IK

Polymorphism overview

Polymorphism: using a type/method/object in different ways

Categories

- universal polymorphism (applicable in *infinitely many*, extensible ways)
  - ◇ parametric polymorphism: generics
  - ◇ subtype polymorphism: inheritance
- ad hoc polymorphism (applicable in specific, *finite* number of ways)
  - ◇ overloading
  - ◇ casting: explicit type conversion

An earlier example

```
public class Receptionist {
  public Time[] readWakeupTimes(String[] fnames) {
    Time[] times = new Time[fnames.length];
    for (int i = 0; i < fnames.length; ++i) {
      try {
        times[i] = readTime(fnames[i]);
      } catch (java.io.IOException e) {
        times[i] = null;   // no-op
        System.err.println("Could not read " + fnames[i]);
      }
    }
    return times; // maybe sort times before returning?
  }
  ...
}
```

Getting rid of null values

```java
public class Receptionist {
  public Time[] readWakeupTimes(String[] fnames) {
    Time[] times = new Time[fnames.length];
    int j = 0;
    for (int i = 0; i < fnames.length; ++i) {
      try {
        times[j] = readTime(fnames[i]);
        ++j;
      } catch (java.io.IOException e) {
        System.err.println("Could not read " + fnames[i]);
      }
    }
    return java.util.Arrays.copyOf(times,j); // possibly
  }                                          //  sorted
  ...
}
```

Advantages and drawbacks of arrays

- Efficient access to elements (indexing)
- Syntactic support in the language (indexing, array literals)
- Length is fixed at construction
  - ◇ Extension with, or removal of, elements is expensive
  - ◇ Requires the creation of a new array, and copying
- Some methods have unexpected (wrong) implementations
  - ◇ This makes arrays incompatible with some data structures

Alternative: java.util.ArrayList

convenient standard library, similar inner workings

```
String[] names = { "Tim",
                   "Jerry" };



names[0] = "Tom";
String mouse = names[1];

String[] trio = new String[3];
trio[0] = names[0];
trio[1] = names[1];
trio[2] = "Spike";
names = trio;
```

```
ArrayList<String> names =
          new ArrayList<>();
names.add("Tim");
names.add("Jerry");

names.set(0, "Tom");
String mouse = names.get(1);

names.add("Spike");
```

Example updated

```
public class Receptionist {
  ...
  public ArrayList<Time> readWakeupTimes(String[] fnames) {
    ArrayList<Time> times = new ArrayList<Time>();
    for (int i = 0; i < fnames.length; ++i) {
      try {
          times.add(readTime(fnames[i]));
      } catch (java.io.IOException e) {
          System.err.println("Could not read " + fnames[i]);
      }
    }
    return times; // possibly sort before returning
  }
}
```

Parametrized type

```
ArrayList<Time> times
```

```
Time[] times
Time times[]
```

Generic class

Not exactly this way, but almost...

```
package java.util;
public class ArrayList<T> {
    public ArrayList() { ... }
    public T get(int index) { ... }
    public void set(int index, T item) { ... }
    public void add(T item) { ... }
    public T remove(int index) { ... }
    ...
}
```

Type parameter is provided when used

```
import java.util.ArrayList;

...

ArrayList<Time> times;
ArrayList<String> names = new ArrayList<String>();
ArrayList<String> namez = new ArrayList<>();
```

Generic method

```
import java.util.*;
class Main {
    public static <T> void reverse(T[] array) {
        int lo = 0, hi = array.length-1;
        while(lo < hi) {
            T tmp = array[hi];
            array[hi] = array[lo];
            array[lo] = tmp;
            ++lo; --hi;
        }
    }
    public static void main(String[] args) {
        reverse(args);
        System.out.println(Arrays.toString(args));
    }
}
```

Parametric polymorphism

- The same code works for many type parameters
  - ◇ What sort of code can take type parameters?
    - ▶ types (classes)
    - ▶ methods
- Code parametrized with any (reference) type

Type parameter

Primitive types not allowed!

```
ArrayList<int> numbers     // compilation error
```

Type parameter

Primitive types not allowed!

```
ArrayList<int> numbers     // compilation error
```

Reference types are OK!

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(Integer.valueOf(7));
Integer seven = numbers.get(0);
```

Type parameter

Primitive types not allowed!

```
ArrayList<int> numbers     // compilation error
```

Reference types are OK!

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(Integer.valueOf(7));
Integer seven = numbers.get(0);
```

Automatic conversion from/to primitive values

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(42);              // auto-boxing:   int -> Integer
int n42 = numbers.get(1);     // auto-unboxing: Integer -> int
```

Auto-(un)boxing

- Automatic two-way conversion
- Between primitive type and its wrapper class

```
Integer ref = 42;
int pri = ref;

Integer sum = ref + pri;
```

```
Integer ref = Integer.valueOf(42);
int     pri = ref.intValue();
Integer sum = Integer.valueOf(
              ref.intValue()
              + pri          );
```

Data structures in java.util

Sequence

```
ArrayList<String> colors = new ArrayList<>();
colors.add("red"); colors.add("white"); colors.add("red");
String third = colors.get(2);
```

Data structures in java.util

Sequence

```
ArrayList<String> colors = new ArrayList<>();
colors.add("red"); colors.add("white"); colors.add("red");
String third = colors.get(2);
```

Set

```
HashSet<String> colors = new HashSet<>();
colors.add("red"); colors.add("white"); colors.add("red");
int two = colors.size();
```

Data structures in java.util

Sequence

```
ArrayList<String> colors = new ArrayList<>();
colors.add("red"); colors.add("white"); colors.add("red");
String third = colors.get(2);
```

Set

```
HashSet<String> colors = new HashSet<>();
colors.add("red"); colors.add("white"); colors.add("red");
int two = colors.size();
```

Mapping

```
HashMap<String,String> colors = new HashMap<>();
colors.put("red", "piros"); colors.put("white", "fehér");
String whiteHu = colors.get("white");
```

Generic class

```
public class ArrayList<T> {
    public ArrayList() { ... }
    public T get(int index) { ... }
    public void set(int index, T item) { ... }
    public void add(T item) { ... }
    public T remove(int index) { ... }
    ...
}
```

Implementation of generic class

```
public class ArrayList<T> {
    private T[] data;
    private int size = 0;
    ...
    public T get(int index) {
        if (index < size) return data[index];
        else throw new IndexOutOfBoundsException();
    }
    ...
}
```

Implementation of generic class

```
import java.util.Arrays;
public class ArrayList<T> {
    private T[] data;
    private int size = 0;
    ...
    public void add(T item) {
        if (size == data.length) {
            data = Arrays.copyOf(data,data.length+1);
        }
        data[size] = item;
        ++size;
    }
    ...
}
```

Allocation attempt: compilation error

```
public class ArrayList<T> {
    private T[] data;
    private int size = 0;
    ...
    public ArrayList() { this(256); }
    public ArrayList(int initialCapacity) {
        data = new T[initialCapacity];
    }
    ...
}
```

```
ArrayList.java:6: error: generic array creation
        data = new T[initialCapacity];
```

Type erasure

- Type parameter: used during static type checking
- Target code: independent of type parameter
  - ◇ Haskell is similar
  - ◇ C++ *template* is different
- Compatibility with generic-less code
- Type parameter cannot be used run-time

The target code *can be considered* like this

```
public class ArrayList {
    private Object[] data;
    ...
    public ArrayList() { ... }
    public Object get(int index) { ... }
    public void set(int index, Object item) { ... }
    public void add(Object item) { ... }
    public Object remove(int index) { ... }
    ...
}
```

Compatibility: raw type

```
import java.util.ArrayList;
...
ArrayList<String> parametrized = new ArrayList<>();
parametrized.add("Romeo");
parametrized.add(12);        // compilation error
String s = parametrized.get(0);
```

Compatibility: raw type

```java
import java.util.ArrayList;
...
ArrayList<String> parametrized = new ArrayList<>();
parametrized.add("Romeo");
parametrized.add(12);       // compilation error
String s = parametrized.get(0);

ArrayList raw = new ArrayList();
raw.add("Romeo");
raw.add(12);
Object o = raw.get(0);
```

Allocation: still not fixed

```
public class ArrayList<T> {
    private T[] data;
    private int size = 0;
    ...
    public ArrayList() { this(256); }
    public ArrayList(int initialCapacity) {
        data = new Object[initialCapacity];
    }
    ...
}
```

```
ArrayList.java:6: error: incompatible types:
                  Object[] cannot be converted to T[]
        data = new Object[initialCapacity];
               ^
  where T is a type-variable:
```

Allocation – already valid

```
public class ArrayList<T> {
    private T[] data;
    private int size = 0;
    ...
    public ArrayList() { this(256); }
    public ArrayList(int initialCapacity) {
        data = (T[])new Object[initialCapacity];
    }
    ...
}
```

```
javac ArrayList.java
```

```
Note: ArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Allocation – already valid, but still not perfect...

```java
public class ArrayList<T> {
    private T[] data;
    private int size = 0;
    ...
    public ArrayList() { this(256); }
    public ArrayList(int initialCapacity) {
        data = (T[])new Object[initialCapacity];
    }
    ...
}
```

```
javac -Xlint:unchecked ArrayList.java
```

```
ArrayList.java:6: warning: [unchecked] unchecked cast
  required: T[]         found: Object[]
```

Casting somewhere else?

```
public class ArrayList<T> {
    private Object[] data;
    private int size = 0;
    ...
    public T get(int index) {
        if (index < size) return (T)data[index];
        else throw new IndexOutOfBoundsException();
    }
    ...
}
```

```
javac -Xlint:unchecked ArrayList.java
```

```
ArrayList.java:10: warning: [unchecked] unchecked cast
  required: T          found: Object
```

Warning-free

```
public class ArrayList<T> {
    private Object[] data;
    private int size = 0;
    ...
    @SuppressWarnings("unchecked")
    public T get(int index) {
        if (index < size) return (T)data[index];
        else throw new IndexOutOfBoundsException();
    }
    ...
}
```

Inheritance

```
class A extends B { ... }
```

- A class is defined in terms of another
  - ◇ Only their difference is to be given: A Δ B
  - ◇ Reuse of code

Inheritance

```
class A extends B { ... }
```

- A class is defined in terms of another
  - ◇ Only their difference is to be given: A Δ B
  - ◇ Reuse of code
- Class `A` is the *child class* of `B`, the *parent class*

Inheritance

```
class A extends B { ... }
```

- A class is defined in terms of another
  - Only their difference is to be given: A Δ B
  - Reuse of code
- Class `A` is the *child class* of `B`, the *parent class*
- Transitively:
  - subclass, derived class
  - super class, base class

Inheritance

```
class A extends B { ... }
```

- A class is defined in terms of another
  - ◇ Only their difference is to be given: A Δ B
  - ◇ Reuse of code
- Class `A` is the *child class* of `B`, the *parent class*
- Transitively:
  - ◇ subclass, derived class
  - ◇ super class, base class
- No circularity!

Example

```
public class Time {
  private int hour, min;    // initialized to 00:00
  public int getHour() { ... }
  public int getMin() { ... }
  public void setHour(int hour) { ... }
  public void setMin(int min) { ... }
  public void aMinPassed() { ... }
}
```

Example

```
public class Time {
  private int hour, min;     // initialized to 00:00
  public int getHour() { ... }
  public int getMin() { ... }
  public void setHour(int hour) { ... }
  public void setMin(int min) { ... }
  public void aMinPassed() { ... }
}
```

```
public class ExactTime extends Time {
  private int sec;           // initialized to 00
  public int getSec() { ... }
  public void setSec(int sec) { ... }
  public boolean earlierThan(ExactTime that) { ... }
}
```

Implicit parent class

```
public class Time extends java.lang.Object {
  private int hour, min;    // initialized to 00:00
  public int getHour() { ... }
  public int getMin() { ... }
  public void setHour(int hour) { ... }
  public void setMin(int min) { ... }
  public void aMinPassed() { ... }
}
```

```
public class ExactTime extends Time {
  private int sec;          // initialized to 00
  public int getSec() { ... }
  public void setSec(int sec) { ... }
  public boolean earlierThan(ExactTime that) { ... }
}
```

java.lang.Object

Base class of every class!

```
package java.lang;
public class Object {
  public Object() { ... }

  public String toString() { ... }
  public int hashCode() { ... }
  public boolean equals(Object that) { ... }
  ...
}
```

Constructors are not inherited

```java
public class Time {
  private int hour, min;
  public Time(int hour, int min) {
    if (hour < 0 || hour > 23 || min < 0 || min > 59)
      throw new IllegalArgumentException();
    this.hour = hour;
    this.min = min;
  }
  ...
}
```

Constructors are not inherited

```
public class Time {
  private int hour, min;
  public Time(int hour, int min) {
    if (hour < 0 || hour > 23 || min < 0 || min > 59)
      throw new IllegalArgumentException();
    this.hour = hour;
    this.min = min;
  }
  ...
}
```

```
public class ExactTime extends Time {
  private int sec;
```

Constructors are not inherited

```
public class Time {
  private int hour, min;
  public Time(int hour, int min) {
    if (hour < 0 || hour > 23 || min < 0 || min > 59)
      throw new IllegalArgumentException();
    this.hour = hour;
    this.min = min;
  }
  ...
}
```

```
public class ExactTime extends Time {
  private int sec;
  public ExactTime(int hour, int min, int sec) { ? }
}
```

The child class needs a constructor!

```java
public class Time {
  private int hour, min;
  public Time(int hour, int min) { ... }
  ...
}
```

```java
public class ExactTime extends Time {
  private int sec;
  public ExactTime(int hour, int min, int sec) {
```

The child class needs a constructor!

```
public class Time {
  private int hour, min;
  public Time(int hour, int min) { ... }
  ...
}
```

```
public class ExactTime extends Time {
  private int sec;
  public ExactTime(int hour, int min, int sec) {
    super(hour, min); // must call parent's constructor
    if (sec < 0 || sec > 59)
      throw new IllegalArgumentException();
    this.sec = sec;
  }
}
```

Calling the `super(...)` constructor

- A constructor in the parent class
- For initializing inherited fields
- Must be the first statement!

Calling the super(...) constructor

- A constructor in the parent class
- For initializing inherited fields
- Must be the first statement!

Erroneous!!!

```
public class ExactTime extends Time {
  private int sec;
  public ExactTime(int hour, int min, int sec) {
    if (sec < 0 || sec > 59)
      throw new IllegalArgumentException();
    super(hour,min);
    this.sec = sec;
  }
}
```

Why is this correct? Missing super?!

```
public class Time extends Object {
  private int hour, min;
  public Time(int hour, int min) {
    if (hour < 0 || hour > 23 || min < 0 || min > 59)
      throw new IllegalArgumentException();
    this.hour = hour;
    this.min = min;
  }
  ...
}
```

Implicit super() call

```
public class Time extends Object {
  private int hour, min;
  public Time(int hour, int min) {
    super();
    if (...) throw ...;
    this.hour = hour;
    this.min = min;
  }
  ...
}
```

```
package java.lang;
public class Object {
  public Object() { ... }
...
```

Implicit parent class, implicit constructor, implicit super

```
class A {}
```

Implicit parent class, implicit constructor, implicit super

```java
class A {}
```

```java
class A extends java.lang.Object {
  A() {
    super();
  }
}
```

Constructors in a class

- One or more explicit constructors
- Default constructor

Constructor body

1st statement

- Explicit `this` call
- Explicit `super` call
- Implicit (automatically generated) `super()` call (no-arg!)

Rest of the body

No calls using `this` or `super`!

Interesting error

Seems OK, but...

```
class Base {
  Base(int n) {}
}

class Sub extends Base {}
```

Meaning

```
class Base extends Object {
  Base(int n) {
    super();
  }
}

class Sub extends Base {
  Sub() { super(); }
}
```

A class defined with inheritance

- Members of parent class are inherited
- Can be extended with new members (Java: `extends`)
- Inherited instance methods can be redefined
  - ... and redeclared

Overriding instance methods

redefinition, overriding

```
package java.lang;
public class Object {
   ...
  public String toString() {...} //java.lang.Object@4f324b5c
}
```

Overriding instance methods

redefinition, overriding

```
package java.lang;
public class Object {
   ...
  public String toString() {...} //java.lang.Object@4f324b5c
}
```

```
public class Time {
   ...
  public String toString() {
    return hour + ":" + min; // 8:5
  }
}
```

Slightly better

redefinition, overriding

```
package java.lang;
public class Object {
  ...
  public String toString() {...} //java.lang.Object@4f324b5c
}
```

```
public class Time {
  ...
  public String toString() {           // 8:05
    return String.format("%1$d:%2$02d", hour, min);
  }
}
```

With the optional (recommended) @Override annotation

redefinition, overriding

```
package java.lang;
public class Object {
  ...
  public String toString() {...} //java.lang.Object@4f324b5c
}
```

```
public class Time {
  ...
  @Override
  public String toString() {          // 8:05
    return String.format("%1$d:%2$02d", hour, min);
  }
}
```

Calling super.toString()

```
package java.lang;                    // java.lang.Object@4f324b5c
public class Object {... public String toString() {...} ... }
```

Calling super.toString()

```
package java.lang;                    // java.lang.Object@4f324b5c
public class Object {... public String toString() {...} ... }

public class Time {
  ...
  @Override public String toString() {    // 8:05
    return String.format("%1$d:%2$02d", hour, min);
  }
}
```

Calling super.toString()

```
package java.lang;                       // java.lang.Object@4f324b5c
public class Object {... public String toString() {...} ... }

public class Time {
  ...
  @Override public String toString() {    // 8:05
    return String.format("%1$d:%2$02d", hour, min);
  }
}
public class ExactTime extends Time {
  ...
  @Override public String toString() {    // 8:05:17
    return String.format("%1:%2$02d", super.toString(), sec);
  }
}
```

Overloading versus overriding

```
package java.lang;
public final class Integer extends Number {
  ...
  public static    int parseInt(String str)          { ... }
  ...
  public @Override String toString()                 { ... }
  public static    String toString(int i)            { ... }
  public static    String toString(int i, int radix) { ... }
  ...
}
```

Differences

Overloading

- Methods/ctors with same name but different parameters
- Introduced method may overload inherited one (Java-specific)
- Compiler selects method/ctor based on actual parameters

Overriding

- Override a method defined in a base class
- Same name, same parameters
  - Same method
  - A method may have multiple implementations
- The most specific implementation is chosen run-time

Static versus dynamic binding

System.out

```
public void println(   int value) { ... }
public void println(Object value) { ... } //value.toString()
...
```

```
System.out.println(7);                     // 7
System.out.println("Samurai");             // Samurai
System.out.println(new Time(21,30));       // 21:30
```

Multiple inheritance

- A type inherits from multiple types directly
- In Java: multiple interfaces
- MI raises problems

Examples

OK

```
package java.util;
public class Scanner implements Closeable, Iterator<String>
{ ... }
```

OK

```
interface PoliceCar extends Car, Emergency { ... }
```

Erroneous

```
class PoliceCar extends Car, Emergency { ... }
```

Hypothetically

```
class Base1 {
    int x;
    void setX( int x ){ this.x = x; }
    ...
}

class Base2 {
    int x;
    void setX( int x ){ this.x = x; }
    ...
}

class Sub extends Base1, Base2 { ... }
```

Hypothetically: diamond-shaped inheritance

```
class Base0 {
    int x;
    void setX( int x ){ this.x = x; }
    ...
}

class Base1 extends Base0 { ... }

class Base2 extends Base0 { ... }

class Sub extends Base1, Base2 { ... }
```

Differences between classes and interfaces

- Classes can be instantiated
  - ◇ `abstract class`?
- Classes support only single inheritence
  - ◇ `final class`?
- Classes may contain instance fields
  - ◇ In interfaces: `public static final`

Multiple inheritance from interfaces

```
interface Base1 {
    abstract void setX( int x );
    ...
}

interface Base2 {
    abstract void setX( int x );
    ...
}

class Sub implements Base1, Base2 {
    void setX( int x ){ ... }
    ...
}
```

Two aspects of inheritance

- Code reuse
- Subtyping

Inheritance gives rise to subtyping

$$A \,\Delta\, B \;\Rightarrow\; A <: B$$

Inheritance gives rise to subtyping

$$A \mathrel{\Delta} B \;\Rightarrow\; A <: B$$

```
public class ExactTime extends Time { ... }
```

- `ExactTime` has everything `Time` has
- Whatever you can do with `Time`, you can do with `ExactTime`
- `ExactTime` $<:$ `Time`

Inheritance gives rise to subtyping

$$A \,\Delta\, B \;\Rightarrow\; A <: B$$

```
public class ExactTime extends Time { ... }
```

- `ExactTime` has everything `Time` has
- Whatever you can do with `Time`, you can do with `ExactTime`
- `ExactTime` $<:$ `Time`
- $\forall T\,\mathrm{class}:\; T <:$ `java.lang.Object`

Subtyping

```java
public class Time {
    ...
    public void aMinutePassed() { ... }
    public boolean sameHourAs(Time that) { ... }
}
```

```java
public class ExactTime extends Time {
    ...
    public boolean isEarlierThan(ExactTime that) { ... }
}
```

```java
ExactTime time = new ExactTime();          // 0:00:00
time.aMinutePassed();                      // 0:01:00
time.sameHourAs(new ExactTime())           // true
```

LSP: Liskov's Substitution Principle

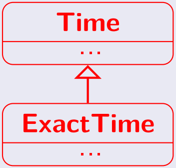
Type A is the subtype of (base-)type B, if instances of A can be used wherever instances of B are used without a problem.

Polymorphic references

```
public class Time {
  ...
  public void aMinutePassed() { ... }
  public boolean sameHourAs(Time that) { ... }
}
```

```
public class ExactTime extends Time {
  ...
  public boolean isEarlierThan(ExactTime that) { ... }
}

ExactTime time1 = new ExactTime();
Time      time2 = new ExactTime(); // upcast
time2.sameHourAs(time1)
```



Time
...
ExactTime
...

Static and dynamic type

Static type: *declared* type of variable / parameter / return value / ...

- Follows from program text
- Does not change during program execution
- Is used by the compiler during static type checking

```
Time time
```

Static and dynamic type

Static type: *declared* type of variable / parameter / return value / ...

- Follows from program text
- Does not change during program execution
- Is used by the compiler during static type checking

```
Time time
```

Dynamic type: *actual* type of variable / parameter / return value / ...

- May change during program execution
- Is meaningful only during run time
- Is the subtype of the static type

```
time = ... ? new ExactTime() : new Time()
```

Overriding

```
package java.lang;                        // java.lang.Object@4f324b5c
public class Object {... public String toString() {...} ... }
```

Overriding

```
package java.lang;                      // java.lang.Object@4f324b5c
public class Object {... public String toString() {...} ... }

public class Time {
  ...
  @Override public String toString() {    // 8:05
    return String.format("%1$d:%2$02d", hour, min);
  }
}
```

Overriding

```
package java.lang;                 // java.lang.Object@4f324b5c
public class Object {... public String toString() {...} ... }

public class Time {
  ...
  @Override public String toString() {    // 8:05
    return String.format("%1$d:%2$02d", hour, min);
  }
}
public class ExactTime extends Time {
  ...
  @Override public String toString() {    // 8:05:17
    return String.format("%1:%2$02d", super.toString(), sec);
  }
}
```

Overloading versus overriding

Overloading

- Same name, different formal parameters
- Inherited methods can be overloaded (in Java!)
- Compiler selects method based on actual parameters

Overriding

- An instance method defined in a base class
- Same name, same formal parameters (same signature)
  - ◇ Same method
  - ◇ Multiple implementations
- **The “most specific” implementation is selected at run time**

Dynamic binding (or *late binding*)

```
ExactTime e = new ExactTime();
Time      t = e;
Object    o = t;

System.out.println(e.toString());     // 0:00:00
System.out.println(t.toString());     // 0:00:00
System.out.println(o.toString());     // 0:00:00
```

At the invocation of an instance method, the implementation that matches the dynamic type of the privileged parameter best will be selected.

The role of static and dynamic type

Static type

What can we do with an expression?

- Static type checking

```
Object o = new Time();
o.setHour(8);              // compilation error
```

Dynamic type

- Which implementation of an instance method?

```
Object o = new Time();
System.out.println(o);   // select toString()
                         // implementation
```

- Dynamic type checking

Inheritance example

```java
package company.hr;
public class Employee {
  String name;
  int basicSalary;
  java.time.ZonedDateTime startDate;
  ...
}
```

Inheritance example

```java
package company.hr;
public class Employee {
  String name;
  int basicSalary;
  java.time.ZonedDateTime startDate;
  ...
}
```

```java
package company.hr;
import java.util.*;
public class Manager extends Employee {
  final HashSet<Employee> workers = new HashSet<>();
  ...
}
```

Parent class

```
package company.hr;
import java.time.ZonedDateTime;
import static java.time.temporal.ChronoUnit.YEARS;
public class Employee {
  ...
  private ZonedDateTime startDate;
  public int yearsInService() {
    return (int) startDate.until(ZonedDateTime.now(), YEARS);
  }
  private static int bonusPerYearInService = 0;
  public int bonus() {
    return yearsInService() * bonusPerYearInService;
  }
}
```

Child class

```
package company.hr;
import java.util.*;
public class Manager extends Employee {
  // inherited: startDate, yearsInService() ...
  ...
  private final HashSet<Employee> workers = new HashSet<>();
  public void addWorker(Employee worker) {
    workers.add(worker);
  }
  private static int bonusPerWorker = 0;
  @Override public int bonus() {
    return workers.size() * bonusPerWorker + super.bonus();
  }
}
```

Dynamic binding also in inherited methods

```java
public class Employee {
  ...
  private int basicSalary;
  public int bonus() {
    return yearsInService() * bonusPerYearInService;
  }
  public int salary() { return basicSalary + bonus(); }
}
```

```java
public class Manager extends Employee {
  ...
  @Override public int bonus() {
    return workers.size()*bonusPerWorker + super.bonus();
  }
}
```

Dynamic binding also in inherited methods

```
Employee jack = new Employee("Jack", 10000);
Employee pete = new Employee("Pete", 12000);
Manager eve = new Manager("Eve", 12000);
Manager joe = new Manager("Joe", 12000);
eve.addWorker(jack);
joe.addWorker(eve);        // polymorphic formal parameter
joe.addWorker(pete);

Employee[] company = {joe, eve, jack, pete};//<-heterogeneous
                                            // data structure
int totalSalaryCosts = 0;
for(Employee e: company) {
  totalSalaryCosts += e.salary();
}
```

Dynamic binding

At the invocation of an instance method, the implementation that matches the dynamic type of the privileged parameter best will be selected.

Fields and static methods cannot be overriden

```
class Base {
  int field = 3;
  int iMethod() { return field; }
  static int sMethod() { return 3; }
}

class Sub extends Base {
  int field = 33;                              // hiding
  static int sMethod() { return 33; }          // hiding
}

Sub sub = new Sub();        Base base = sub;
sub.sMethod()  ==  33       base.sMethod()  ==  3
sub.field      ==  33       base.field      ==  3
sub.iMethod()  ==  3        base.iMethod()  ==  3
```

Inheritance ⇒ subtyping

`class A implements I`

A Δ_{ci} I ⇒ A <: I

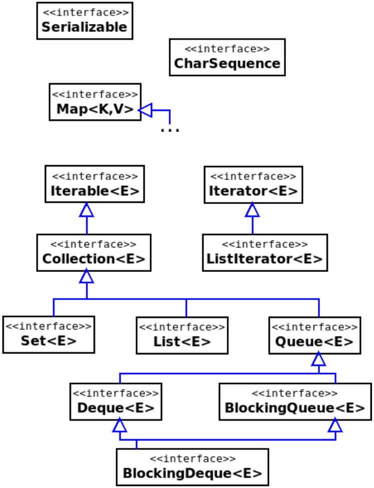
`class A extends B`

A Δ_{c} B ⇒ A <: B

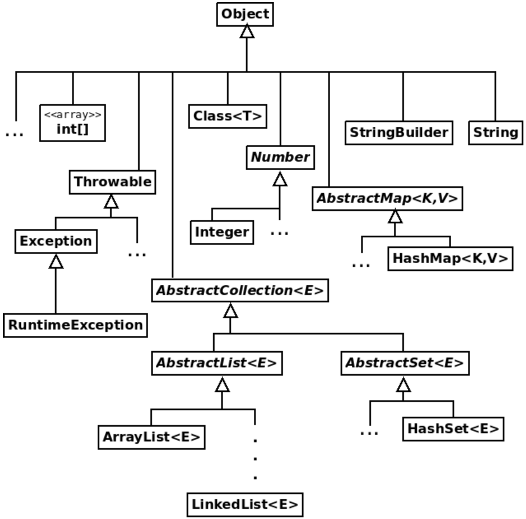
`interface I extends J`

I Δ_{i} J ⇒ I <: J

Interface hierarchy in Java API (fragment)


<<interface>> Serializable
<<interface>> CharSequence
<<interface>> Map<K,V>
...
<<interface>> Iterable<E>
<<interface>> Iterator<E>
<<interface>> Collection<E>
<<interface>> ListIterator<E>
<<interface>> Set<E>
<<interface>> List<E>
<<interface>> Queue<E>
<<interface>> Deque<E>
<<interface>> BlockingQueue<E>
<<interface>> BlockingDeque<E>

Class hierarchy in Java API (fragment)



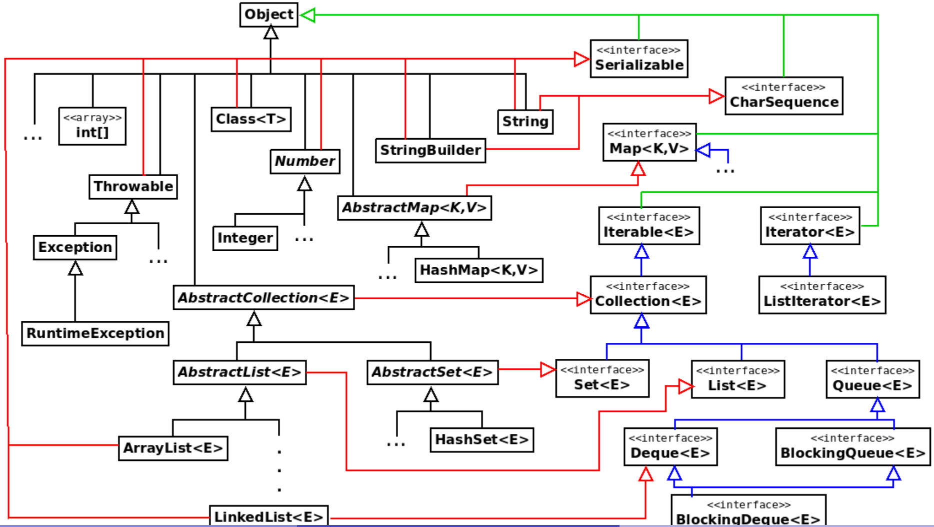
java.lang.Object

Each class is derived from it (except itself)

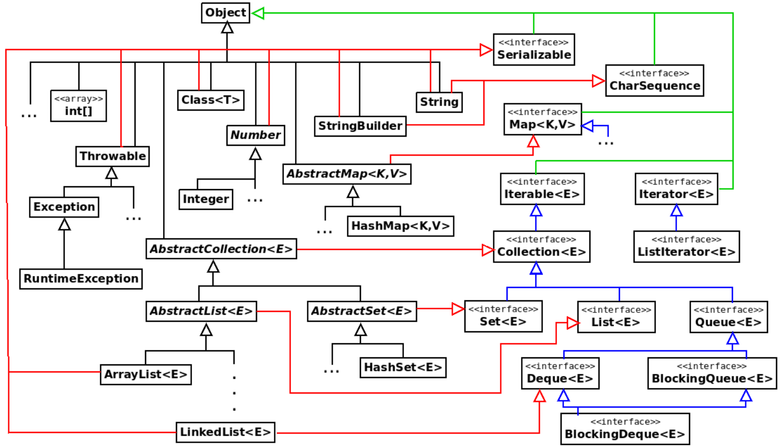
```
package java.lang;
public class Object {
    public Object(){ ... }
    public String toString(){ ... }
    public int hashCode(){ ... }
    public boolean equals( Object that ){ ... }
    public Class getClass(){ ... }
    ...
}
```

Hierarchy of reference types in Java API (fragment)

DAG: directed acyclic graph



Hierarchy of types in Java API (fragment)



```
boolean
char
byte
short
int
long
float
double
```

Subtype relation

$$<: \quad = \quad (\Delta_c \cup \Delta_i \cup \Delta_{ci} \cup \Delta_o)^*$$

- Δ_o means: each reference type is derived from `java.lang.Object`
- ϱ^* means: reflexive, transitive closure of relation ϱ
  - ◇ If `A` ϱ `B` then `A` ϱ^* `B`
  - ◇ Reflexive closure: `A` ϱ^* `A`
  - ◇ Tranzitive closure: if `A` ϱ^* `B` and `B` ϱ^* `C` then `A` ϱ^* `C`

This is a partial order over types (RAT)!

The dynamic type is the subtype of the static type

If `A` $<:$ `B` then

- `B v = new A();` is correct
- `void m( B p )...` allows `m(new A())`
- `A a; B b; ... b = a;` is correct

Subtype polymorphism

A code base developed for base types can be reused with subtypes

- Object of more specific types can be used instead those of more general types
- The same code base can be used with many types: polymorphism

Reusability!

Specialization

- The subtype is “capable of everything” the base type is capable of
- The subtype can be specialized
- This is the *is-a* relation
  - ◇ `Car` *is-a* `Vehicle`
  - ◇ `Boat` *is-a* `Vehicle`
- Human thinking, object-oriented modelling

Multiple inheritance of concepts

- A concept may specialize multiple concepts
  - ◇ `PoliceCar` *is-a* `Car` **and** *is-a* `EmergencyVehicle`
  - ◇ `FireBoat` *is-a* `Boat` **and** *is-a* `EmergencyVehicle`
- Complex conceptual modeling in Java: `interface`

Multiple inheritance of code?

- Inheritance of code: classes
    - ◇ only single inheritance

Multiple inheritance of code?

- Inheritance of code: classes
  - ◇ only single inheritance
- From `interfaces`
  - ◇ multiple inheritance
  - ◇ inheritance of code is limited: methods with `default` implementation

Hierarchy of exception classes

java.lang.Throwable

- java.lang.Exception
  - ◇ java.sql.SQLException
  - ◇ java.io.IOException
    - ▶ java.io.FileNotFoundException
  - ◇ ...
  - ◇ programmer defined exceptions come typically here
  - ◇ java.lang.RuntimeException
    - ▶ java.lang.NullPointerException
    - ▶ java.lang.ArrayIndexOutOfBoundsException
    - ▶ java.lang.IllegalArgumentException
    - ▶ ...
- java.lang.Error
  - ◇ java.lang.VirtualMachineError
  - ◇ ...

Unchecked exceptions

- `java.lang.RuntimeException` and its subclasses
- `java.lang.Error` and its subclasses

May be required to handle in critical applications!

catch-branches

```
try {
    ...
} catch( FileNotFoundException e ){
    ...
} catch( EOFException e ){
    ...
} // java.net.SocketException is not handled
```

Specific and general catch-branches

```
try {
    ...
} catch( FileNotFoundException e ){
    ...
} catch( EOFException e ){
    ...
} catch( IOException e ){  // every other IOException
    ...
}
```

Compilation error: unreachable code

```
try {
    ...
} catch( FileNotFoundException e ){
    ...
} catch( IOException e ){  // every other IOException
    ...
} catch( EOFException e ){ // wrong order!
    ...
}
```

Motivating example

```
package java.lang;
public interface CharSequence {
  int length();
  char charAt( int index );
  ...
}
```

Implementing classes

- java.lang.String
- java.lang.StringBuilder
- java.lang.StringBuffer
- ...

Let's define lexicographic ordering!

```
static boolean less(CharSequence left, CharSequence right){..
```

Subtype polymorphism is suitable here

```
static boolean less(CharSequence left, CharSequence right) {
   ...
}
```

```
less( "cool", "hot" )

StringBuilder sb1 = new StringBuilder(); ...
StringBuilder sb2 = new StringBuilder(); ...
less( sb1, sb2 )

less( "cool", sb1 )
```

Subtype polymorphism is not enough here

```java
static boolean less(CharSequence left, CharSequence right) {
  ...
}

static CharSequence min(CharSequence left,
                        CharSequence right) {
  return less(left,right) ? left : right;
}
```

Subtype polymorphism is not enough here

```
static boolean less(CharSequence left, CharSequence right) {
  ...
}

static CharSequence min(CharSequence left,
                        CharSequence right) {
  return less(left,right) ? left : right;
}
```

OK

```
CharSequence cs = min( "cool", "hot" );
```

Compilation error

```
String str = min( "cool", "hot" );
```

Parametric polymorphism

```
static boolean less(CharSequence left, CharSequence right) {
  ...
}

static <T> T min( T left, T right ){
  return less(left,right) ? left : right;
}
```

Compilation error: `less`

Bounded universal quantification

```
static boolean less(CharSequence left, CharSequence right) {
  ...
}
static <T extends CharSequence> T min(T left, T right) {
  return less(left,right) ? left : right;
}
```

Bounded universal quantification

```
static boolean less(CharSequence left, CharSequence right) {
  ...
}
static <T extends CharSequence> T min(T left, T right) {
  return less(left,right) ? left : right;
}
```

```
String str = min( "cool", "hot" );
StringBuilder sb = min( new StringBuilder(),
                        new StringBuilder() );
CharSequence cs = min( "cool", new StringBuilder() );
```

Bounded universal quantification

- constrained genericity
- bounded parametric polymorphism
- $\forall T$ derived from `CharSequence`, we define the `min` function...
- upper bound

Natural ordering

```
java.util.Arrays.sort(args)
```

Natural ordering

```
java.util.Arrays.sort(args)
```

```
public final class String ... {
  ...
  public int compareTo( String that ){ ... }
}
```

```
public final class Integer ... {
  ...
  public int compareTo( Integer that ){ ... }
}
```

Natural ordering – interface

3-way comparison

```
package java.lang;
public interface Comparable<T> {   // negative: this < that
  int compareTo( T that );         //     zero: this = that
}                                  // positive: this > that
```

Natural ordering – interface

3-way comparison

```
package java.lang;
public interface Comparable<T> {    // negative: this < that
  int compareTo( T that );          //     zero: this = that
}                                   // positive: this > that
```

```
package java.lang;
public final class String implements Comparable<String> {...}
// public int compareTo( String that ){ ... }
```

```
package java.lang;
public final class Integer implements Comparable<Integer>{...}
// public int compareTo( Integer that ){ ... }
```

Define natural ordering for your class

3-way comparison

```
package java.lang;
public interface Comparable<T> {     // negative: this < that
  int compareTo( T that );           //     zero: this = that
}                                    // positive: this > that
```

```
public class Rational implements Comparable<Rational> {
  ...
  public int compareTo( Rational that ){
    /* class invariant: denominator > 0 */
    return numerator * that.denominator -
            that.numerator * denominator;
  }
}
```

Inherit natural ordering

```
public class Rational implements Comparable<Rational> {
  ...
  public int compareTo( Rational that ){
    return numerator * that.denominator -
            that.numerator * denominator;
  }
}
```

```
public class SimpleRational extends Rational { ... }
```

```
(new Rational(3,6)).compareTo(new Rational(5,9))
(new Rational(3,6)).compareTo(new SimpleRational(5,9))
(new SimpleRational(3,6)).compareTo(new SimpleRational(5,9))
(new SimpleRational(3,6)).compareTo(new SimpleRational(5,9))
(new SimpleRational(3,6)).compareTo(new Rational(5,9))
```

Problem with inheritance

A class cannot implement the same generic interface multiple times with different type parameters.

```
public class Time implements Comparable<Time> {
  ...
  public int compareTo( Time that ){ ... }
}
```

Compilation error

```
public class ExactTime extends Time
                       implements Comparable<ExactTime> {
  ...
  public int compareTo( ExactTime that ){ ... }
}
```

Other orderings

```
@FunctionalInterface
public interface Comparator<T> {
  int compare( T left, T right );    // 3-way
}
```

Other orderings

```
@FunctionalInterface
public interface Comparator<T> {
  int compare( T left, T right );    // 3-way
}
```

```
class StringLengthComparator implements Comparator<String> {
  public int compare( String left, String right ){
    return left.length() - right.length();
  }
}
```

```
java.util.Arrays.sort(args, new StringLengthComparator());
```

Other orderings

```
@FunctionalInterface
public interface Comparator<T> {
  int compare( T left, T right );    // 3-way
}
```

```
class StringLengthComparator implements Comparator<String> {
  public int compare( String left, String right ){
    return left.length() - right.length();
  }
}
```

```
java.util.Arrays.sort(args, new StringLengthComparator());
```

```
java.util.Arrays.sort(args, (a,b) -> a.length()-b.length());
```

Sorting

Public operations offered by the `java.util.Arrays` class:

```
static <T> void parallelSort(T[] a, Comparator<? super T> cmp)
```

- `cmp`: there exist a type S which is a base type of T, and `cmp` can compare S objects
  - existential quantification
  - lower bound

Sorting

Public operations offered by the `java.util.Arrays` class:

```
static <T> void parallelSort(T[] a, Comparator<? super T> cmp)
```

- cmp: there exist a type S which is a base type of T, and cmp can compare S objects
  - ◇ existential quantification
  - ◇ lower bound

```
static <T extends Comparable<? super T>>
                                        void parallelSort(T[] a)
```

- Type T must have a base type which supports natural ordering

Subtype relation for parametrized types

```
public class ArrayList<T> ... implements List<T> ...
```

∀ T: `ArrayList<T>` <: `List<T>`

- `ArrayList<String>` <: `List<String>`
- `ArrayList<Integer>` <: `List<Integer>`

Subtypes of type parameter?

Rule

`List<Integer>` $\not<:$ `List<Object>`

Subtypes of type parameter?

Rule

List<Integer> ≮: List<Object>

Indirect assumption: List<Integer> <: List<Object>

```
List<Integer> nums = new ArrayList<Integer>();
nums.add( 42 );                   // Integer.valueOf(42)
List<Object> things = nums;       // use assumption
things.add( "forty-two" );        // String <: Object
Integer n = nums.get(1);          // contradiction!
```

Arrays obey to a weaker rule

Java allows `Integer[]` $<:$ `Object[]`

Arrays obey to a weaker rule

Java allows `Integer[]` <: `Object[]`

Parametrized types

```
List<Integer> nums = new ArrayList<Integer>();
nums.add( 42 );                   // Integer.valueOf(42)
List<Object> things = nums;       // compilation error
things.add( "forty-two" );        // String <: Object
Integer n = nums.get(1);          // would cause problem
```

Arrays obey to a weaker rule

Java allows `Integer[] <: Object[]`

Parametrized types

```
List<Integer> nums = new ArrayList<Integer>();
nums.add( 42 );                  // Integer.valueOf(42)
List<Object> things = nums;      // compilation error
things.add( "forty-two" );       // String <: Object
Integer n = nums.get(1);         // would cause problem
```

Array types

```
Integer[] nums = new Integer[2];
nums[0] =  42 ;                  // Integer.valueOf(42)
Object[] things = nums;          // allowed
things[1] = "forty-two" ;        // ArrayStoreException
Integer n = nums[1];             // would cause problem
```

A class defined with inheritance

- Members of parent class are inherited
- Can be extended with new members (Java: `extends`)
- Inherited instance methods can be redefined
  - ◇ ... and **redeclared**

Example: cloning

```
package java.lang;
public interface Cloneable {}
```

```
package java.lang;
public class CloneNotSupportedException extends Exception{...}
```

```
package java.lang;
public class Object {
  public boolean equals( Object that ){ return this==that; }
  ...
  protected Object clone() throws CloneNotSupportedException{
    if( this instanceof Cloneable ) return /* shallow copy */
    else throw new CloneNotSupportedException();
  }
}
```

Shallow copy – 1st try

```
public class Time implements Cloneable {
  private int hour, minute;
  public Time( int hour, int minute ){ ... }
  public void setHour( int hour ){ ... }
  ...
  @Override public String toString(){ ... }
  @Override public int hashCode(){ return 60*hour+minute; }
  @Override public boolean equals( Object that ){
    if(that != null && getClass().equals(that.getClass())){
      Time t = (Time)that;
      return hour == t.hour && minute == t.minute;
    } else return false;
  }
}
```

Inconvenient!

```
package java.lang;
public class Object {
  ...
  protected Object clone() throws CloneNotSupportedException..
}
```

Cannot be called from everywhere!

```
public class Time implements Cloneable {
  ...
  public static void main( String[] args ){
    Time t = new Time(12,30);
    try { Object o = t.clone(); }
    catch( CloneNotSupportedException e ){ assert false; }
  }
}
```

Redeclaration solves the problems

```
public class Time implements Cloneable {
  private int hour, minute;
  ...
  @Override public Time clone(){
    try { return (Time)super.clone(); }
    catch( CloneNotSupportedException e ){
      assert false; return null;
    }
  }
}
```

Legal overriding

- Visibility can be widened
- Return type can be narrowed
- Declared thrown checked exceptions can be narrowed

Rules of cloning

`implements Cloneable`

- Make `clone()` public
  - ◇ overriding
  - ◇ redeclaration

Rules of cloning

`implements Cloneable`

- Make `clone()` public
  - ◇ overriding
  - ◇ redeclaration
- Always use `super.clone()`!
  - ◇ preserves dynamic type
  - ◇ inherited `clone()` will return object with correct dynamic type

Rules of cloning

`implements Cloneable`

- Make `clone()` public
    - ◇ overriding
    - ◇ redeclaration
- Always use `super.clone()`!
    - ◇ preserves dynamic type
    - ◇ inherited `clone()` will return object with correct dynamic type
- Shallow copy: implementation in `Object` can be used
    - ◇ Deep(er) copy: requires additional work
    - ◇ immutable field need not be cloned

Deep copy

```
public class Interval implements Cloneable {
  private Time from, to;
  public void setFrom( Time from ){
    this.from.setHour( from.getHour() );
    this.from.setMinute( from.getMinute() );
  }
  @Override public Interval clone(){
    try { Interval that = (Interval)super.clone();
          that.from = that.from.clone();
          that.to = that.to.clone();
          return that;
    } catch( CloneNotSupportedException e ){
          assert false; return null;
    }
  }
  ...
```

Type conversions between primitive types

Automatic type conversion (transitive)

- `byte` → `short` → `int` → `long`
- `long` → `float`
- `float` → `double`
- `char` → `int`
- `byte b = 42;` and `short s = 42;` and `char c = 42;`

Explicit type cast

```
int i = 42;
short s = (short)i;
```

Puzzle 3: Long Division (Bloch & Gafter: Java Puzzlers)

```java
public class LongDivision {
  public static void main(String[] args) {
    final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
    final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
    System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
  }
}
```

Wrapper classes

Implicitly imported (`java.lang`), immutable classes

- `java.lang.Boolean` – `boolean`
- `java.lang.Character` – `char`
- `java.lang.Byte` – `byte`
- `java.lang.Short` – `short`
- `java.lang.Integer` – `int`
- `java.lang.Long` – `long`
- `java.lang.Float` – `float`
- `java.lang.Double` – `double`

The interface of java.lang.Integer (fragment)

```
static int MAX_VALUE    // 2^31-1
static int MIN_VALUE    // -2^31

static int compare(int x, int y)    // 3-way comparison
static int max(int x, int y)
static int min(int x, int y)
static int parseInt(String str [, int radix])
static String toString(int i [, int radix])
static Integer valueOf(int i)

int compareTo(Integer that)         // 3-way comparison
int intValue()
```

Auto-(un)boxing

- Automatic two-way conversion
- Between primitive type and its wrapper class

```
Integer ref = 42;
int pri = ref;

Integer sum = ref + pri;
```

```
Integer ref = Integer.valueOf(42);
int     pri = ref.intValue();
Integer sum = Integer.valueOf(
              ref.intValue()
              + pri          );
```

ELTE
IK

Auto-(un)boxing + generics

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(7);
int seven = numbers.get(0);
```

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(Integer.valueOf(7));
int seven = numbers.get(0).intValue();
```

Computation with integers

```
int n = 10;
int fact = 1;
while (n > 1) {
    fact *= n;
    --n;
}
```

Auto-(un)boxing costs more

```
Integer n = 10;
Integer fact = 1;
while (n > 1) {
    fact *= n;
    --n;
}
```

Auto-(un)boxing costs more

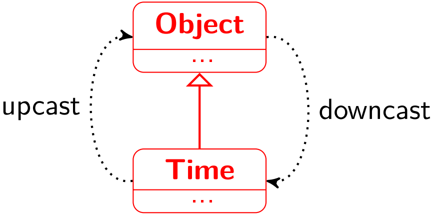
```
Integer n = 10;
Integer fact = 1;
while (n > 1) {
    fact *= n;
    --n;
}
```

Meaning:

```
Integer n = Integer.valueOf(10);
Integer fact = Integer.valueOf(1);
while (n.intValue() > 1) {
    fact = Integer.valueOf(fact.intValue() * n.intValue());
    n = Integer.valueOf(n.intValue() - 1);
}
```

Conversions on reference types

- Automatic (upcast) – subtyping
- Explicit (downcast) – type-cast operator



Type cast (downcast)

- The static type of the expression “`(Time)o`” is `Time`

Type cast (downcast)

- The static type of the expression “(Time)o” is Time
- If the dynamic type of o is Time:

```
Object o = new Time(3,20);
o.aMinutePassed();             // compilation error
((Time)o).aMinutePassed();     // compiles. works.
```

Type cast (downcast)

- The static type of the expression “(Time)o” is Time
- If the dynamic type of o is Time:

```
Object o = new Time(3,20);
o.aMinutePassed();             // compilation error
((Time)o).aMinutePassed();     // compiles. works.
```

- If not, ClassCastException is thrown

```
Object o = "Twenty passed three";
o.aMinutePassed();             // compilation error
((Time)o).aMinutePassed();     // run-time error
```

Dynamic type checking

- During run-time, based on dynamic type
- More precise than static type checking
    - ◇ Dynamic types can be subtypes
- Flexibility
- Safety: only when explicitly requested (type cast)

instanceof operator

```
Object o = new ExactTime(3,20,0);
...
if (o instanceof Time) {
    ((Time)o).aMinutePassed();
}
```

- Dynamic type of given expression is a subtype of given type

instanceof operator

```
Object o = new ExactTime(3,20,0);
...
if (o instanceof Time) {
    ((Time)o).aMinutePassed();
}
```

- Dynamic type of given expression is a subtype of given type
- Static type and given type have to be related

```
"apple" instanceof Integer    // compilation error
```

instanceof operator

```
Object o = new ExactTime(3,20,0);
...
if (o instanceof Time) {
    ((Time)o).aMinutePassed();
}
```

- Dynamic type of given expression is a subtype of given type
- Static type and given type have to be related

```
"apple" instanceof Integer    // compilation error
```

- `null` yields `false`

Representation of dynamic types during run-time

- objects of class `java.lang.Class`
- can be accessed run-time

```
Object o  = new Time(17,25);
Class  c  = o.getClass();      // Time.class
Class  cc = c.getClass();      // Class.class
```

Inheritance – subtyping

- `class A extends B` …
- $A <: B$
- $\forall T: \quad T <:$ `java.lang.Object`

Automatic “conversion” to base type (upcast)

```
String str = "Java";
Object o = str;  // OK
str = o;         // compilation error
```

Type case to subtype (downcast)

```
String str = "Java";
Object o = str;  // OK
str = (String)o; // OK, dynamic type checking
```

ClassCastException

```
String str = "Java";
Object o = str;
Integer i = (Integer)o;
```

A value belongs to a type (subtyping allowed)

```java
String str = "Java";
Object o = str;
Integer i = (o instanceof Integer) ? (Integer)o : null;
```

Exact match of dynamic types

```
String str = "Java";
Object o = str;
Integer i = o.getClass().equals(Integer.class) ?
                (Integer)o : null;
```