

**GIUSEPPE TURINI**

**CS-102: COMPUTING AND ALGORITHMS 2**

**LESSON 01**

**INTRO, OOP, AND JAVA**

# HIGHLIGHTS

## Java Fundamentals

Primitive Types, Classes, References, Object Equality and Cloning, Immutables  
Modifiers, Method Signature, Overloading and Overriding Methods  
Interfaces, Exceptions, Error Reporting, Packages, and Java Garbage Collector

## Object-Oriented Programming (OOP) Principles

**Encapsulation:** Information Hiding, Modularity, and Violations

**Abstraction:** Procedural and Data Abstraction

**Inheritance:** Subclasses and Superclasses, and Casting Objects

**Polymorphism:** Static Binding, Dynamic Binding, and Abstract Classes

## Java Generics

Generic Types and Methods, and Bounded Type Parameters  
Type Inference, and Wildcards

## Java Extras

Boxing, and the Java Garbage Collector Scheduling

# STUDY GUIDE

## STUDY MATERIAL

- This slides.

## SELECTED EXERCISES

- Review the content of the prerequisite course (CS-101).

## ADDITIONAL RESOURCES

- **“Object-Oriented Data Structures Using Java (4<sup>th</sup> Ed.)”, chap. 1.**
- “Data Abstraction and Problem Solving with Java (3<sup>rd</sup> Ed.)”, chap. 1-2, chap. 4.
- “Java Illuminated (5<sup>th</sup> Ed.)”, chap. 1-3, chap. 7, chap. 10-11.

# JAVA FUNDAMENTALS - PRIMITIVE TYPES

**Java is statically-typed:** Variables must be declared (name+type) before being used.

**Java has 8 primitive data types:** Predefined types named by reserved keywords.

PRIMITIVE DATA TYPES

Data Type	Description	Default Value
byte and short	8-bit signed two's complement integer in $[-2^7, 2^7-1]$ , and 16-bit signed two's complement integer in $[-2^{15}, 2^{15}-1]$ , respectively.	0 and 0
int	32-bit signed two's complement integer in $[-2^{31}, 2^{31}-1]$ . In Java SE 8 and later, int can represent an unsigned 32-bit integer in $[0, 2^{32}-1]$ , using the Integer class.	0
long	64-bit signed two's complement integer in $[-2^{63}, 2^{63}-1]$ . In Java SE 8 and later, long can represent an unsigned 64-bit long in $[0, 2^{64}-1]$ , using the Long class.	0L
float and double	Single-precision 32-bit IEEE 754 floating point, and double-precision 64-bit IEEE 754 floating point, respectively.	0.0f and 0.0d
char	Single 16-bit Unicode character in $['\u0000'$ (or 0), $'\uffff'$ (or 65,535)].	$'\u0000'$
boolean	Type with only 2 possible values: true and false; and its size depends on the Java Virtual Machine.	false
object reference	Address indicating where an object is stored in memory. Usually its size equals the native pointer size (32 bits for a 32 bit JVM).	null

**See:** [docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes)

# JAVA FUNDAMENTALS - CLASSES 1

## DESCRIPTION OF A JAVA CLASS

In Java, a **class is a new data type** whose instances are objects, and including the following class members:

- **data fields** (should almost always be private),
- **methods**.

## ACCESS CONTROL OF A JAVA CLASS

- A **class member** with no access modifier is **package-private**, otherwise it can be configured as **private**, **public**, or **protected**.
- A **class** with no access modifier is **package-private**, otherwise it can be configured as **public**.

**See:** [docs.oracle.com/javase/tutorial/java/javaoo/accesscontrol](https://docs.oracle.com/javase/tutorial/java/javaoo/accesscontrol)

**A program or module that uses a class is usually referred as a client of a class.**

# JAVA FUNDAMENTALS - CLASSES 2

## JAVA CLASS CONSTRUCTORS

The **class constructor** is a method that creates and initializes new instances (i.e. objects) of a class. It has the same name as the class, no return type, and may have parameters. The class constructor allocates the memory required to store an object. A class can have more than one constructor. A constructor with no parameters is usually called the **default constructor**.

**If a class has no constructor,  
the Java compiler generates automatically a default constructor for it!**

**See:** [docs.oracle.com/javase/tutorial/java/javaoo/constructors](https://docs.oracle.com/javase/tutorial/java/javaoo/constructors)

# JAVA FUNDAMENTALS - REFERENCES 1

## REFERENCE VARIABLES

A variable that refers to an object of a class, is actually a reference to that object. A **reference variable** stores the location (i.e. **memory address**) of an object.

```
Integer intRef; // Declaration of a reference variable intRef.  
intRef = new Integer(5); // Instantiate an Integer obj and assign its ref to intRef.
```

When a reference variable is used as a data field of a class, its default value is **null**. But, if it is used as a local variable in a method, it does not have a default value.

**Note:** An object of a class does not come into existence until you call one of the class constructors using the **new** operator.

# JAVA FUNDAMENTALS - REFERENCES 2

## EXAMPLE OF A REFERENCE VARIABLE

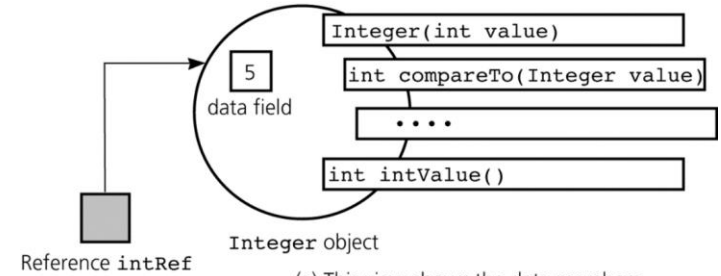
```
// Declare a reference variable intRef,  
// referencing an object of type Integer.  
Integer intRef;
```

```
// Instantiate an Integer object, and  
// assign its reference to intRef.  
intRef = new Integer(5);
```

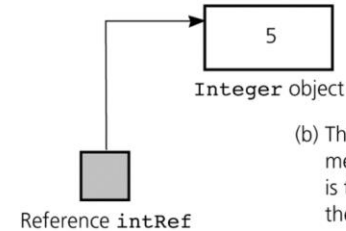
**Note:** The primitive type **int** is different than the class **Integer**! In fact, the **Integer** class wraps a value of the primitive type **int** in an object.

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/integer](https://docs.oracle.com/javase/8/docs/api/java/lang/integer)

**See:** [docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes)



(a) This view shows the data members and methods for the object.



(b) This view only shows the data members for simplicity. This is the view used throughout the text.



# JAVA FUNDAMENTALS - REFERENCES 3

## OPERATIONS ON REFERENCE VARIABLES A

If you try to use a reference variable that does not currently reference any object (i.e. with value **null**), the **exception java.lang.NullPointerException** will be thrown at runtime! Trying to use a reference variable not initialized will cause a compiler error!

When one reference variable is assigned to another reference variable, both references then refer to the same object

```
Integer p, q; // p and q do not reference any object.  
p = new Integer(6); // p now references an Integer object of value 6.  
q = p; // Now both p and q reference the same Integer object of value 6.
```

**Note:** If an object is not referenced by any variable is marked for garbage collection!

# JAVA FUNDAMENTALS - REFERENCES 4

## OPERATIONS ON REFERENCE VARIABLES B

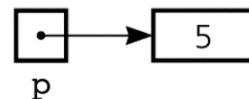
- Declaration of reference variables **p** and **q**, does not initialize of their references.
- Allocation of an **Integer** object (**5**), and storage of its reference into **p**.
- Allocation of another **Integer** object (**6**), and storage of its reference again into **p**. This overwrite **p**, dereferencing object (**5**) that is marked for garbage collection (**gray**).

- Assign **p** to **q**, so now both reference the same **Integer** object (**6**).

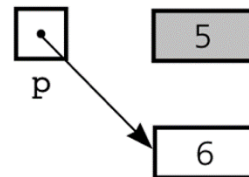
(a) `Integer p;`  
`Integer q;`



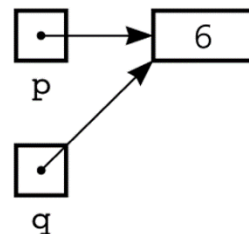
(b) `p = new Integer(5);`



(c) `p = new Integer(6);`



(d) `q = p;`

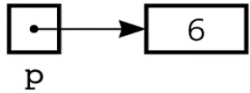


# JAVA FUNDAMENTALS - REFERENCES 5

## OPERATIONS ON REFERENCE VARIABLES C

- e. Allocation of an **Integer** object (**9**), and storage of its reference into **q**. Since **p** still references the same object (**6**), that object is not marked for garbage collection.
- f. Assignment of a **null** reference to **p**, this dereferences at the same time object (**6**) that is marked for garbage collection (**gray**).
- g. Assign **p** to **q**, so now: both variables have a **null** value, and object (**9**) is dereferenced and so it is marked for garbage collection.

(e) `q = new Integer(9);`




The diagram shows two variables, p and q, each in a box with a dot. p has an arrow pointing to a box containing the number 6. q has an arrow pointing to a box containing the number 9.

(f) `p = null;`



The diagram shows two variables, p and q, each in a box with a dot. p's box is crossed out with a diagonal line, indicating it is null. q has an arrow pointing to a box containing the number 9.

(g) `q = p;`



The diagram shows two variables, p and q, each in a box with a dot. Both p's box and q's box are crossed out with a diagonal line, indicating they are both null.

# JAVA FUNDAMENTALS - REFERENCES 6

## ARRAYS AND REFERENCES

An array of objects is actually an array of references to the objects.

```
Integer[] scores = new Integer[30]; // An array of 30 references to Integer objects.
```

**Note:** Remember that you need to instantiate objects for each array item!

```
scores[0] = new Integer(7); // Initialize reference stored into array element 0.
```

**See:** [docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays)

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/integer](https://docs.oracle.com/javase/8/docs/api/java/lang/integer)

# JAVA FUNDAMENTALS - REFERENCES 7

## EQUALITY BETWEEN REFERENCES

Equality operators `==` and `!=` compare the values of reference variables, not the values of the referenced objects.

To compare objects field by field, use the method **equals** (**java.lang.Object** class).

**See:** [docs.oracle.com/javase/tutorial/java/nutsandbolts/op2](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2)

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/object](https://docs.oracle.com/javase/8/docs/api/java/lang/object)

# JAVA FUNDAMENTALS - REFERENCES 8

## PASSING OBJECTS TO METHODS

**Formal Parameter:** A variable as found in the function definition.

**Actual Argument:** The actual input passed to a function.

When a method is called and has formal parameters that are reference variables, then **the reference value of the actual argument is copied to the formal parameter.**

```
void f( Integer fp ) { ... } // fp is a formal parameter of method f.
```

```
Integer aa = new Integer(7); // aa is an actual argument of the following call to f.  
f( aa ); // Note: in this call to f both aa and fp reference the same Integer object (7)!
```

```
// Note: using the new operator with a formal parameter can produce unexpected results!  
void f( Integer fp ) { fp = new Integer(9); ... } // Warning: unexpected results!
```

# JAVA FUNDAMENTALS - OBJECT EQUALITY

## OBJECT EQUALITY

Objects should be not compared directly with the `==` operator. To compare objects you should use or implement the method **equals**.

**equals** is a method of the **Object** class, and in its default implementation compares 2 objects checking their references.

However, a **custom implementation of the method equals** can be developed (via **method overriding**) to check objects data for equality.

**See:** [docs.oracle.com/javase/tutorial/java/nutsandbolts/op2](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2)

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/object](https://docs.oracle.com/javase/8/docs/api/java/lang/object)

**See:** [docs.oracle.com/javase/tutorial/java/iandi/override](https://docs.oracle.com/javase/tutorial/java/iandi/override)

# JAVA FUNDAMENTALS - OBJECT CLONING 1

## OBJECT CLONING

Object cloning is the creation of an exact copy of an existing object in memory.

In Java, the **clone()** method in the **Object** class is used for cloning. This method creates an (shallow) copy of an object and returns the reference of that clone.

Not all the objects in Java are eligible for cloning process. The objects which implement **Cloneable** interface are only eligible for cloning process.

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/object](https://docs.oracle.com/javase/8/docs/api/java/lang/object)

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/cloneable)



# JAVA FUNDAMENTALS - OBJECT CLONING 2

Both **shallow copy** and **deep copy** are related to this cloning process.

## SHALLOW COPY

A shallow copy of an object stores exact copies of all the fields of the original object. If the original object has any references as fields, then these references are copied. In this case the shallow copy is not fully disjoint from the original object. The default version of **clone()** method creates a shallow copy of an object.

## DEEP COPY

A deep copy of an object is like a shallow copy, but if the original object has any references to other objects as fields, then copy of those objects are also created. This means that a deep copy is fully independent from the original object.

# JAVA FUNDAMENTALS - IMMUTABLES

## MUTABLE AND IMMUTABLE OBJECTS

An object is **immutable** if its state cannot change after it is constructed. Max reliance on immutable objects is accepted as a sound strategy for creating reliable code.

Immutable objects are useful in **concurrent applications**. They cannot change state, so they cannot be corrupted by thread interference or found in an inconsistent state.

Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place. The **impact of object creation is often overestimated**, and can be offset by some of the efficiencies associated with immutable objects.

**See:** [docs.oracle.com/javase/tutorial/essential/concurrency/immutable](https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable)

# JAVA FUNDAMENTALS - MODIFIERS 1

## ACCESS MODIFIERS

Access modifiers determine if other classes can use a field or method. There are two levels of access control:

- **at class level:** public, package-private (no modifier).
- **at field-method level:** public, private, protected, package-private (no modifier).

The following table shows the access to members permitted by each modifier.

ACCESS LEVELS				
Modifier	Class	Package	Subclass	World
public	yes	yes	yes	yes
protected	yes	yes	yes	no
no modifier	yes	yes	no	no
private	yes	no	no	no

**See:** [docs.oracle.com/javase/tutorial/java/javaoo/accesscontrol](https://docs.oracle.com/javase/tutorial/java/javaoo/accesscontrol)

# JAVA FUNDAMENTALS - MODIFIERS 2

## NON-ACCESS MODIFIERS

- **static**: for creating class methods and variables.
- **final**: for finalizing implementations of classes-methods-variables.
- **abstract**: for creating abstract classes and methods.
- **synchronized** and **volatile**: which are used for threads.

## STATIC CLASS VARIABLES

Fields that have the **static** modifier in their declaration are called **class variables**.

They are associated with the class, rather than with any object.

Every class instance shares the class variable, that is in one fixed location in memory.

A class variables can also be manipulated without creating an instance of the class.

**See:** [docs.oracle.com/javase/tutorial/java/javaoo/classvars](https://docs.oracle.com/javase/tutorial/java/javaoo/classvars)

**See:** [docs.oracle.com/javase/tutorial/java/landl/final](https://docs.oracle.com/javase/tutorial/java/landl/final)

# JAVA FUNDAMENTALS - MODIFIERS 3

## STATIC CLASS METHODS

Java supports **static methods** (having **static** modifier in their declarations) should be invoked with the class name, without the need for creating an instance of the class.

Not all combinations of instance and class variables and methods are allowed:

- **instance methods** can access **instance variables/methods** directly;
- **instance methods** can access **class variables/methods** directly;
- **class methods** can access **class variables/methods** directly;
- **class methods** cannot access **instance variables/methods** directly (they must use an object reference);
- **class methods** cannot use the **this** keyword (there is no instance to refer to).

**See:** [docs.oracle.com/javase/tutorial/java/javaoo/classvars](https://docs.oracle.com/javase/tutorial/java/javaoo/classvars)

**See:** [docs.oracle.com/javase/tutorial/java/landl/final](https://docs.oracle.com/javase/tutorial/java/landl/final)

# JAVA FUNDAMENTALS - MODIFIERS 4

## CONSTANTS - STATIC AND FINAL VARIABLES

The **static** modifier, together with the **final** modifier, is also used to define **constants**. The **final** modifier indicates that the value of this field cannot change.

**Note:** Re-assigning a final variable is a compile-time error.

**Note:** Constants names should be in uppercase (words separated by an underscore).

**Note:** If a **primitive type** or a **String** is defined as a constant and the value is known at compile-time (**compile-time constant**), the compiler replaces the constant name everywhere in the code with its value. In this case, if the constant value changes, you will need to recompile any classes that use this constant to get the updated value.

**See:** [docs.oracle.com/javase/tutorial/java/javaoo/classvars](https://docs.oracle.com/javase/tutorial/java/javaoo/classvars)

**See:** [docs.oracle.com/javase/tutorial/java/landl/final](https://docs.oracle.com/javase/tutorial/java/landl/final)

# JAVA FUNDAMENTALS - MODIFIERS 5

## FINAL METHODS

Declaring a method **final** means that **it cannot be overridden by subclasses**. The **Object** class includes several methods that are **final**.

**Note:** Declare a method **final** if its body is critical to the consistent state of the object.

**Note:** Methods called from constructors should be **final**, to avoid their redefinition by a subclass that could lead to undesirable results.

## FINAL CLASSES

Declaring a class **final** means that the class **cannot be subclassed**. For example, this is useful when creating immutable classes like the **String** class.

**See:** [docs.oracle.com/javase/tutorial/java/landl/final](https://docs.oracle.com/javase/tutorial/java/landl/final)

# JAVA FUNDAMENTALS - METHOD SIGNATURE

In Java, method declarations have six components, in order:

1. **Modifiers:** public, private, etc.
2. **Return Type:** the type of the value returned, or void if no value is returned.
3. **Method Name:** check naming conventions for methods.
4. **Formal Parameters:** a comma-separated parameter list in parentheses.
5. **Exceptions:** a list of exceptions the method can throw.
6. **Method Body:** the method code enclosed between braces.

**Method Signature:** the method name together with its parameter types.

```
public int func( double par1, String par2 ) { ... } // Method declaration.
```

```
func( double, String ) // Method signature.
```

**See:** [docs.oracle.com/javase/tutorial/java/javaoo/methods](https://docs.oracle.com/javase/tutorial/java/javaoo/methods)



# JAVA FUNDAMENTALS - METHOD OVERLOADING

**Java supports overloading methods:** methods within a class can have the same name if they have different parameter lists.

- Overloaded methods are differentiated by the number and type of input args.
- A class cannot have 2 methods with same method signature.
- The Java compiler does not consider return type when differentiating methods.

**Note:** Overloaded methods should be used sparingly (they make code less readable).

```
public int func( double par1, String par2 ) { ... } // Method declaration.
```

```
public int func( float par1 ) { ... } // Method declaration (overloading).
```

```
public double func( double par1, String par2 ) { ... } // Error: same signature.
```

**See:** [docs.oracle.com/javase/tutorial/java/javaoo/methods](https://docs.oracle.com/javase/tutorial/java/javaoo/methods)

# JAVA FUNDAMENTALS - OVERRIDING METHODS 1

## OVERRIDING INSTANCE METHODS

An instance method in a subclass with the same **signature and return type** as an instance method in the superclass overrides the method in the superclass.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed.

**Note:** An overriding method can also **return a subtype** of the type returned by the overridden method. This subtype is called a **covariant return type**.

**Note:** When overriding a method, you might want to use the **@Override** annotation that instructs the compiler that you intend to override a method in the superclass.

**See:** [docs.oracle.com/javase/tutorial/java/landl/override](https://docs.oracle.com/javase/tutorial/java/landl/override)

**See:** [docs.oracle.com/javase/tutorial/java/annotations](https://docs.oracle.com/javase/tutorial/java/annotations)

# JAVA FUNDAMENTALS - OVERRIDING METHODS 2

## OVERRIDING (HIDING) STATIC METHODS

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass.

## HIDING STATIC METHODS VS OVERRIDING INSTANCE METHODS

- The overridden instance method invoked is the one in the subclass.
- The hidden static method invoked depends on the class it is invoked from.

## MODIFIERS AND OVERRIDING

- **Access Modifiers:** Can allow more, not less, access than the overridden method.
- **Static Modifier:** You will get a compile-time error if you change an instance method in the superclass to a static method in the subclass, and vice versa.

**See:** [docs.oracle.com/javase/tutorial/java/land/override](https://docs.oracle.com/javase/tutorial/java/land/override)

# JAVA FUNDAMENTALS - OVERRIDING METHODS 3

## HIDING A STATIC METHOD VS OVERRIDING AN INSTANCE METHOD

```
public class Animal {  
    public static void classMethod() { System.out.print( "Static method in Animal." ); }  
    public void instanceMethod() { System.out.print( "Instance method in Animal." ); }  
}
```

```
public class Cat extends Animal {  
    public static void classMethod() { System.out.print( "Static method in Cat." ); }  
    public void instanceMethod() { System.out.print( "Instance method in Cat." ); }  
    public static void main( String[] args ) {  
        Cat myCat = new Cat(); Animal myAnimal = myCat;  
        Animal.testClassMethod(); // OUTPUT: Static method in Animal.  
        myAnimal.testInstanceMethod(); // OUTPUT: Instance method in Cat.  
    }  
}
```

**See:** [docs.oracle.com/javase/tutorial/java/landl/override](https://docs.oracle.com/javase/tutorial/java/landl/override)

# JAVA FUNDAMENTALS - INTERFACES 1

## JAVA INTERFACE A

An interface specifies methods and constants but supplies no implementation (used to specify a common behavior). Java API includes many interfaces (e.g. **java.lang.Comparable**).

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/comparable)

### A class that implements an interface must:

- include an **implements** clause, and
- provide implementations of the methods of the interface.

### To define an interface:

- use the keyword **interface** instead of **class** in the header, and
- provide only method specifications and constants in the interface definition.

# JAVA FUNDAMENTALS - INTERFACES 2

## JAVA INTERFACE B

### Defining an interface:

```
public interface MyInterface {  
    public final int f1 = 0;  
    public void method1();  
    public int method2( int a, int b ); }
```

### Defining a class that implements an interface:

```
public class MyClass implements MyInterface {  
    public void method1() {}  
    public int method2( int a, int b ) { return a + b; } }
```

# JAVA FUNDAMENTALS - INTERFACES 3

## JAVA INTERFACE C

### OBJECT COMPARISON AND INTERFACES

We can only compare objects that are “comparable”. In Java this means that their class has to implement the **java.lang.Comparable interface** containing the method **compareTo** that the object class has to implement to perform the comparison.

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/comparable)

```
// The SimpleSphere class implements the interface Comparable (Java API).
public class SimpleSphere implements java.lang.Comparable<Object> {
    public int compareTo( Object rhs ) { // Note: "rhs" stands for right-hand side.
        SimpleSphere other = (SimpleSphere) rhs; // Note: possible ClassCastException!
        if( radius == other.getRadius() ) { return 0; } // This is equal to "other".
        else if( radius < other.getRadius() ) { return -1; } // This is less than "other".
        else { return 1; } } // This is greater than "other".
}
```

# JAVA FUNDAMENTALS - EXCEPTIONS 1

An **exception is a mechanism for handling an error during execution**. A method indicates that a runtime error has occurred by "throwing" an exception. In your code, you can handle a runtime error by "catching" the exception thrown by the method causing the error during execution.

## CATCHING EXCEPTIONS

To catch exceptions, code that might throw an exception is enclosed in a **try** block:

```
try { ... }
```

Right after a **try** block, one or more **catch** blocks can be used to handle the error:

```
catch( ExceptionClass e ) { ... }
```



# JAVA FUNDAMENTALS - EXCEPTIONS 2

## TYPES OF EXCEPTIONS: CHECKED EXCEPTIONS

Instances of classes that are subclasses of the **java.lang.Exception** class, these exceptions must be handled locally or explicitly thrown from the method. Are used in situations where the method has encountered a **serious problem**.

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/exception](https://docs.oracle.com/javase/8/docs/api/java/lang/exception)

## TYPES OF EXCEPTIONS: RUNTIME EXCEPTIONS

Instances of classes that are subclasses of the **java.lang.RuntimeException** class, these exception are not required to be caught locally or explicitly thrown. These exceptions are usually used when the **error is not critical**.

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/runtimeexception](https://docs.oracle.com/javase/8/docs/api/java/lang/runtimeexception)

# JAVA FUNDAMENTALS - EXCEPTIONS 3

## THROWING EXCEPTIONS

A **throw** statement is used to throw an exception:

```
throw new MyCustomException( "My custom message." );
```

## DEFINING CUSTOM EXCEPTIONS

To define custom exception classes representing specific runtime error types:

```
import java.lang.RuntimeException;
import java.lang.String;

public class MyCustomException extends RuntimeException {
    public MyCustomException( String s ) { super(s); } // Constructor.
}
```

# JAVA FUNDAMENTALS - ERROR REPORTING

In Java, there are 3 ways to print an exception information, all using methods in the **Throwable** class (the superclass for all exceptions and errors).

- **printStackTrace():** that prints the name and description of the exception object, and its stack trace (the code location where this exception occurred).  
`e.printStackTrace();` // Where e is a reference to an object of type Exception.
- **toString():** that prints only the name and description of this exception object. This method (originally in the **Object** class) is overridden by the **Throwable** class.  
`System.out.println( e.toString() );` // ...
- **getMessage():** that returns the description of this exception object as a **String**.  
`System.out.println( e.getMessage() );` // ...

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/throwable](https://docs.oracle.com/javase/8/docs/api/java/lang/throwable)

# **JAVA FUNDAMENTALS - PACKAGES** 1

**Java packages provide a way to group related classes together.**

**You must use the same name for both  
a Java class and the file that contains that class.**

**You must use the same name for both  
a package and the directory that contains all the classes in that package.**

# JAVA FUNDAMENTALS - PACKAGES 2

To create a package, place a **package** statement before each class in the package:

```
package MyPackage;  
public class MyClass {} // Note: public class available to all clients of the package.
```

To make the class available to the clients of the package, use the keyword **public** (access modifier) before the class declaration. If there is no access modifier, the class is available only to other classes in the package (i.e. **package-private**).

```
package MyPackage;  
class MyClass {} // Note: no access modifier means the class is package-private.
```

**Note:** When a class is publicly available within a package, it can also be used by other classes, even those appearing in other packages.

# JAVA FUNDAMENTALS - PACKAGES 3

A package can contain other packages as well, and in this case the directory hierarchy should be consistent. In fact **the package name consists of the hierarchy of package names, separated by periods.**

```
package MyDrawingPackage.MyShapePackage;  
import java.lang.Object;
```

To use a package in a program we need to use the **import** statement. In particular, you can use the **asterisk notation** to indicate to the Java compiler that you might use any class in that package.

```
import java.io.*;  
import java.io.DataStream;
```

# JAVA FUNDAMENTALS - PACKAGES 4

## CLASSES WITH NO EXPLICIT PACKAGE

If you omit the package declaration from the source file for a class, the class is added to a **default unnamed package**. If all the classes in a group are declared this way, they are all considered to be within this same default unnamed package and hence do not require an **import** statement.

However, **if you are developing a package, and you want to use a class that is contained in the default unnamed package, you will need to import the class**. In this case, since the package has no name, the class name itself is sufficient in the import statement.

```
import MyClass;
```

# JAVA FUNDAMENTALS - THE GARBAGE COLLECTOR

## JAVA GARBAGE COLLECTOR

The **garbage collector (GC)** destroys all objects that a program no longer needs (i.e. references), so there is **no need of class destructors** to deallocate memory.

A GC works in this way:

1. when a program no longer references an object, the **Java Runtime Environment (JRE)** marks it for garbage collection;
2. periodically, the JRE executes a method that free the memory used by these marked objects making this space available for future use;
3. if when an object is destroyed, other tasks beyond memory deallocation are necessary, **you can define a finalize method for a class.**

**See:** [en.wikipedia.org/wiki/garbage\\_collection\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/garbage_collection_(computer_science))



# OOP PRINCIPLES

## OBJECT-ORIENTED PROGRAMMING (OOP)

Object-oriented programming (OOP) views **a program** not as a sequence of actions but **as a collection of components called objects**.

## PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING

- **Encapsulation:** objects combine data and operations.
- **Abstraction:** separate purpose-use from implementation.
- **Inheritance:** classes can inherit from other classes.
- **Polymorphism:** objects can decide operations at run time.

**See:** [docs.oracle.com/javase/tutorial/java/iandi/subclasses](https://docs.oracle.com/javase/tutorial/java/iandi/subclasses)

**See:** [docs.oracle.com/javase/tutorial/java/iandi/polymorphism](https://docs.oracle.com/javase/tutorial/java/iandi/polymorphism)

# OOP PRINCIPLES - ENCAPSULATION 1

## ENCAPSULATION

**The term encapsulation is often used interchangeably with information hiding.**

In programming languages, encapsulation refers to one of these distinct notions:

- A tool to restrict access to some object components (**information hiding**).
- A tool to bundle data with methods operating on that data (**encapsulation**).

***“encapsulation** separates the interface of an abstraction and its implementation.”*

**Grady Booch.** *Object-Oriented Analysis and Design with Applications.* 2007.

**See:** [en.wikipedia.org/wiki/encapsulation\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/encapsulation_(computer_programming))

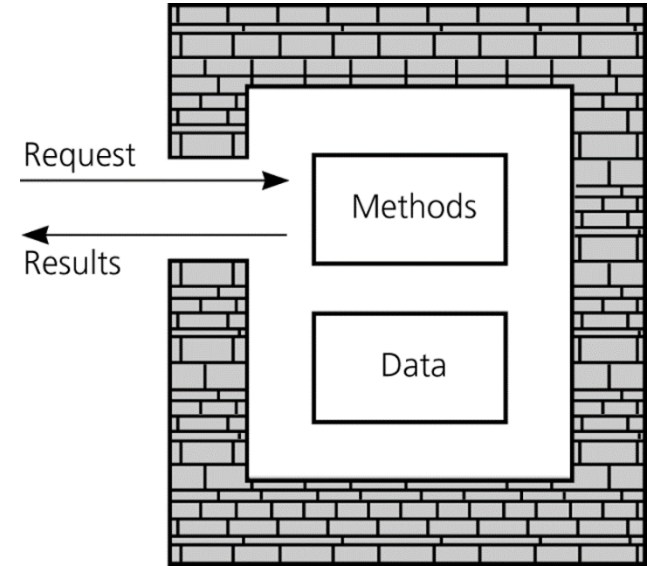
**See:** [en.wikipedia.org/wiki/information\\_hiding](https://en.wikipedia.org/wiki/information_hiding)

# OOP PRINCIPLES - ENCAPSULATION 2

## ENCAPSULATION

The **packing of data and functions** into a single element (i.e. an object).

**Encapsulation combines the data with its operations to form an object.**



**See:** [en.wikipedia.org/wiki/encapsulation\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/encapsulation_(computer_programming))

**See:** [en.wikipedia.org/wiki/information\\_hiding](https://en.wikipedia.org/wiki/information_hiding)

# OOP PRINCIPLES - INFORMATION HIDING 1

## INFORMATION HIDING

Information hiding is the ability to prevent certain aspects of a software component from being accessible to its clients.

In OOP, information hiding reduces software development risk by shifting the code dependency from an uncertain implementation onto a well-defined interface.

Clients of the interface perform operations purely through it; so, if the implementation changes, the clients do not have to change.

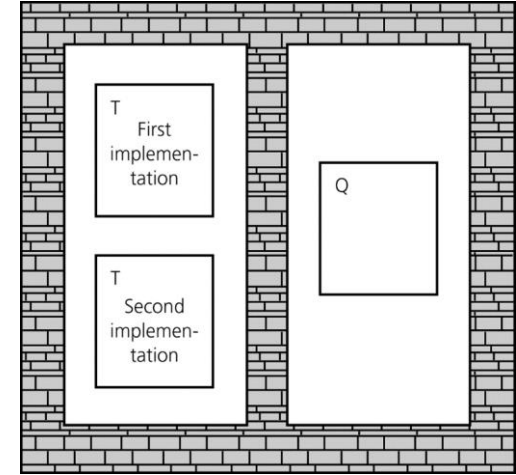
**See:** [en.wikipedia.org/wiki/encapsulation\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/encapsulation_(computer_programming))

**See:** [en.wikipedia.org/wiki/information\\_hiding](https://en.wikipedia.org/wiki/information_hiding)

# OOP PRINCIPLES - INFORMATION HIDING 2

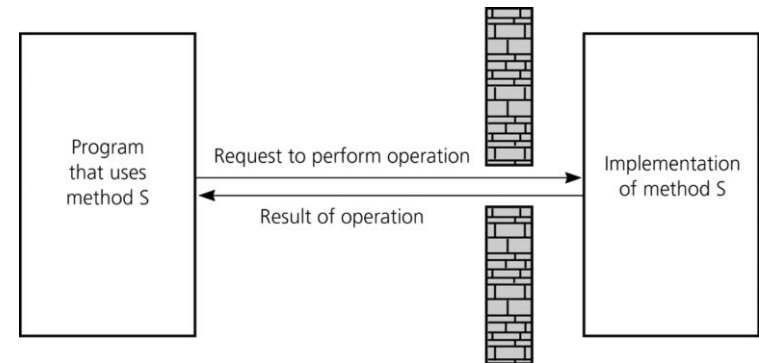
## TOTALLY ISOLATED TASKS

Implementation of task **T** does not affect task **Q**.



## PARTIALLY ISOLATED MODULES

Specifications of methods govern how they interact with each other.



# OOP PRINCIPLES - MODULARITY

## MODULARITY

Keeps the complexity of a large program manageable by systematically controlling the interaction of its components. Isolates errors and eliminates redundancies.

A **modular program** is:

- easier to write,
- easier to read, and
- easier to modify.

**Note:** Generally speaking, modules should be logically separate pieces of your program. In object-oriented programming (OOP), usually modules are defined via classes and their relationships.

# OOP PRINCIPLES - VIOLATING ENCAPSULATION

1

## EXAMPLE VIOLATING DATA ENCAPSULATION A

```
import java.lang.StringBuilder;
public class ExampleViolatingDataEncapsulation {
    static public void main( String[] args ) {
        // TEST 1
        StringBuilder sb1 = new StringBuilder( "ab" );
        StringBuilder sb2 = new StringBuilder( "xyz" );
        // We print the SortedStringPair object to check if sorting is fine.
        SortedStringPair ssp = new SortedStringPair( sb1, sb2 );
        System.out.println( "1: " + ssp );
        // We try to violate the sorting using the class setter.
        ssp.setShorter( new StringBuilder( "abcd" ) );
        System.out.println( "2: " + ssp );
        // Warning: data encapsulation violated by external code using object references!
        sb1.append( 'c' ); sb1.append( 'd' );
        System.out.println( "3: " + ssp );
        // ...
    }
}
```

# OOP PRINCIPLES - VIOLATING ENCAPSULATION

2

## EXAMPLE VIOLATING DATA ENCAPSULATION B

```
// TEST 2
StringBuilder sb3 = new StringBuilder( "ab" );
StringBuilder sb4 = new StringBuilder( "xyz" );
// We print the SortedStringPair object to check if sorting is fine.
SortedStringPairSafe ssps = new SortedStringPairSafe( sb3, sb4 );
System.out.println( "4: " + ssps );
// We try to violate the sorting using the class setter.
ssps.setShorter( new StringBuilder( "abcd" ) );
System.out.println( "5: " + ssps );
// Warning: trying to violate data encapsulation using object references (1)!
sb3.append( 'c' ); sb3.append( 'd' );
System.out.println( "6: " + ssps );
// Warning: trying to violate data encapsulation using object references (2)!
StringBuilder sb5 = ssps.getShorter();
sb5.append( 'c' ); sb5.append( 'd' );
System.out.println( "7: " + ssps ); }
```

}



# OOP PRINCIPLES - VIOLATING ENCAPSULATION

3

## SORTED STRING PAIR A

```
import java.lang.StringBuilder; // Note: String is immutable!
public class SortedStringPair {
    private StringBuilder shorter;
    private StringBuilder longer;
    public SortedStringPair( StringBuilder a, StringBuilder b ) {
        if( a.length() > b.length() ) { shorter = b; longer = a; }
        // Note: if both strings have equal length we don't sort.
        else { shorter = a; longer = b; } }

    // Note: we set shorter only if input argument does not violate the sorting.
    public void setShorter( StringBuilder s ) {
        if( s.length() < longer.length() ) { shorter = s; } }
    public StringBuilder getShorter() { return shorter; }

    // ...
}
```

# OOP PRINCIPLES - VIOLATING ENCAPSULATION

4

## SORTED STRING PAIR B

```
// Note: we set longer only if input argument does not violate the sorting.
public void setLonger( StringBuilder s ) {
    if( s.length() > shorter.length() ) { longer = s; } }
public StringBuilder getLonger() { return longer; }

@Override
public String toString() {
    return "\"" + shorter + "\" is shorter than \"" + longer + "\"; }

}
```

# OOP PRINCIPLES - VIOLATING ENCAPSULATION 5

## SORTED STRING PAIR SAFE A

```
import java.lang.StringBuilder; // Note: String is immutable!
public class SortedStringPairSafe {
    private StringBuilder shorter;
    private StringBuilder longer;
    public SortedStringPairSafe( StringBuilder a, StringBuilder b ) {
        if( a.length() > b.length() ) {
            shorter = new StringBuilder(b);
            longer = new StringBuilder(a); }
        // Note: if both strings have equal length we don't sort.
        else {
            shorter = new StringBuilder(a);
            longer = new StringBuilder(b); }
    }

    // ...
}
```

# OOP PRINCIPLES - VIOLATING ENCAPSULATION

6

## SORTED STRING PAIR SAFE B

```
// Note: we set shorter only if input argument does not violate the sorting.
public void setShorter( StringBuilder s ) {
    if( s.length() < longer.length() ) { shorter = new StringBuilder(s); } }
public StringBuilder getShorter() { return new StringBuilder(shorter); }

// Note: we set longer only if input argument does not violate the sorting.
public void setLonger( StringBuilder s ) {
    if( s.length() > shorter.length() ) { longer = new StringBuilder(s); } }
public StringBuilder getLonger() { return new StringBuilder(longer); }

@Override
public String toString() {
    return "\"" + shorter + "\" is shorter than \"" + longer + "\"; }

}
```

# OOP PRINCIPLES - ABSTRACTION

## PROCEDURAL ABSTRACTION

Separates the purpose and use of a module from its implementation.

*"... any operation that achieves a well-defined effect can be treated by its users as a single entity ..."* **John Daintith**. *A Dictionary of Computing*. 2004.

The **specifications of a module** should:

- detail how the module behaves, and
- identify details that can be hidden within the module.

## DATA ABSTRACTION

Think **what you can do** to a collection of data independently of **how you do** it.

Allows development of a data structure in relative isolation from the rest of the code.

# OOP PRINCIPLES - INHERITANCE 1

**Inheritance:** In Java, classes can derive from other classes, inheriting fields/methods. Inheritance is a mechanism for **code reuse** and to allow **independent extensions**.

**Subclass:** A class derived from another class (aka derived/extended/child class).

**Superclass:** The class from which the subclass is derived (aka base/parent class).

**Single Inheritance:** Every class in Java has 1 and only 1 direct superclass (excepting the class **java.lang.Object**, which has no superclass).

**Note:** A class with no explicit superclass, is implicitly a subclass of **java.lang.Object**.

**Note:** The relationships of classes through inheritance results in a **class hierarchy**.

**See:** [docs.oracle.com/javase/tutorial/java/iandi/subclasses](https://docs.oracle.com/javase/tutorial/java/iandi/subclasses)

# OOP PRINCIPLES - INHERITANCE 2

**Note:** Use **extends** keyword in base class to inherit from a superclass.

**Note:** Fields and methods are inherited by subclasses, but **constructors** are **not**.

**Note:** Use **super** keyword in base class constructor to call the superclass constructor.

**Note:** A subclass does **not** inherit the **private members** of its parent class.

## CASTING OBJECTS A

```
public class Animal { ... }  
public class Cat extends Animal { ... }
```

Therefore: a **Cat** object is also of type **Animal** and of type **java.lang.Object**.

This means: a **Cat** object can be used where **Animal** or **Object** objects are called for.

**Note:** The reverse is not always true. An **Animal** object may (or not) also be a **Cat**.

**See:** [docs.oracle.com/javase/tutorial/java/iandi/subclasses](https://docs.oracle.com/javase/tutorial/java/iandi/subclasses)

# OOP PRINCIPLES - INHERITANCE 3

## CASTING OBJECTS B

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations.

### Implicit Casting:

```
Object obj = new Cat(); // Note: obj is both of type Cat and Object.
```

### Explicit Casting:

```
Object obj = new Cat(); // Note: obj is both of type Cat and Object.  
Cat c = (Cat) obj; // Note: we tell the compiler that obj can be converted to type Cat.
```

The explicit cast inserts a run-time check so the compiler can safely assume a valid type conversion. If the type conversion fails at runtime, an exception will be thrown.

**See:** [docs.oracle.com/javase/tutorial/java/iandi/subclasses](https://docs.oracle.com/javase/tutorial/java/iandi/subclasses)



# OOP PRINCIPLES - INHERITANCE 4

## CASTING OBJECTS C

You can implement a logical test to check the type of an object using the **instanceof** operator. This can save you from a run-time error owing to an improper cast.

```
if( obj instanceof Cat ) { Cat c = (Cat) obj; }
```

Here the **instanceof** operator checks if **obj** stores a reference to an object of type **Cat** so that we can make the explicit cast knowing that no exception will be thrown.

**See:** [docs.oracle.com/javase/tutorial/java/iandi/subclasses](https://docs.oracle.com/javase/tutorial/java/iandi/subclasses)

# OOP PRINCIPLES - POLYMORPHISM 1

**Polymorphism:** A state of having many shapes, the ability to take on different forms.

**Polymorphism (OOP):** The capacity of an OOP language (Java) to process objects of various types and classes through a single uniform interface.

Polymorphism in Java has 2 types:

- compile-time polymorphism (**static binding**); or
- runtime polymorphism (**dynamic binding**).

**Example:** The class **Animal** has a subclass **Cat**. So, any object of type **Cat**, is also of type **Animal**. Therefore, any object of type **Cat** is polymorphic.

**Note:** It is correct to say that (almost) every object in Java is polymorphic in nature, since all Java classes are derived from the **Object** class.

# OOP PRINCIPLES - POLYMORPHISM 2

## COMPILE-TIME POLYMORPHISM (STATIC BINDING)

In Java, compile-time polymorphism is obtained using **method overloading**.

**Method Overloading:** multiple methods in the same class having the same name but different types/order/number of formal parameters.

**Static Binding:** At compile time, the Java compiler knows which method to invoke by checking the **method signatures**.

```
class ExampleOverloading {  
    public int add(int x, int y) { ... } // Method 1.  
    public int add(int x, int y, int z) { ... } // Method 2.  
    public int add(double x, int y) { ... } // Method 3.  
    ...  
}
```

# OOP PRINCIPLES - POLYMORPHISM 3

## RUN-TIME POLYMORPHISM (DYNAMIC BINDING)

In Java, run-time polymorphism is obtained using **method overriding**.

**Method Overriding:** A subclass can provide a specific implementation of a method that is already included in its superclass. When a subclass method has the **same signature and return type** as the superclass method, the subclass method is said to override the method in the superclass.

**Dynamic Binding:** At runtime, Java knows which method to invoke by checking **the type of the object** pointed by the reference (used to call a method).

```
Fruit f = new Apple(); f.print(); // Calls print method in Apple class (subclass).  
f = new Fruit(); f.print(); // Calls print method in Fruit class (superclass).  
// Note: The type or the reference variable f is always Fruit!
```

# OOP PRINCIPLES - ABSTRACT CLASSES 1

**Abstract Class:** A class declared **abstract**. It may or may not include abstract methods. Abstract classes **cannot be instantiated**, but they **can be subclassed**.

```
public abstract class MyAbstractClass { ... }
```

**Abstract Method:** A method declared **abstract** and providing **no implementation**.

```
public abstract void myAbstractMethod();
```

**Note:** A class including abstract methods must be declared abstract.

**Note:** When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

**See:** [docs.oracle.com/javase/tutorial/java/iandi/abstract](https://docs.oracle.com/javase/tutorial/java/iandi/abstract)

# OOP PRINCIPLES - ABSTRACT CLASSES 2

## ABSTRACT CLASSES COMPARED TO INTERFACES

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However:

- **with abstract classes**, you can declare fields that are **not static** and **not final**, and define **public/protected/private** methods (with implementation) ;
- **with interfaces**, all fields are automatically **public, static, and final**, and all methods that you declare/define are **public**;
- additionally, in Java, **you can derive only 1 class**, whether or not it is abstract, whereas **you can implement any number of interfaces**.

**See:** [docs.oracle.com/javase/tutorial/java/iandi/abstract](https://docs.oracle.com/javase/tutorial/java/iandi/abstract)

# OOP PRINCIPLES - ABSTRACT CLASSES 3

## Consider using abstract classes if:

- you want to **share code** among related classes (see **java.util.AbstractMap**);
- classes derived from the abstract class have many common methods/fields, or require access modifiers other than public;
- you want to declare non-static or non-final fields.

## Consider using interfaces if:

- unrelated classes will implement the interface (see **java.lang.Comparable**);
- you want to **specify the behavior** of a data type, but not its implementation;
- you want to take advantage of **multiple inheritance** of type.

**See:** [docs.oracle.com/javase/tutorial/java/iandi/abstract](https://docs.oracle.com/javase/tutorial/java/iandi/abstract)

**See:** [docs.oracle.com/javase/8/docs/api/java/util/abstractmap](https://docs.oracle.com/javase/8/docs/api/java/util/abstractmap)

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/comparable)

# JAVA GENERICS - GENERIC TYPES 1

Generics **increase code stability** by making more bugs detectable at compile-time. Generics enable classes/interfaces to be parameters for classes/interfaces/methods.

**Type Parameters:** Classes or interfaces, type parameters allow code re-use (with different inputs). The inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

- **strong compile-time type check** (compile-time errors are easier to fix than run-time errors);
- **elimination of casts;**
- coding of **generic algorithms** (working on collections of different types).

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)



# JAVA GENERICS - GENERIC TYPES 2

**Generic Type:** A generic type is a generic class/interface parameterized over types.

## Generic Type Example: A Generic Class to Represent Pairs

```
public class GenericPair< T1, T2 > {  
    private T1 item1; // First item of the pair.  
    private T2 item2; // Second item of the pair.  
    public GenericPair( T1 item1, T2 item2 ) { this.item1 = item1; this.item2 = item2; }  
    public void setItem1( T1 item1 ) { this.item1 = item1; }  
    public T1 getItem1() { return this.item1; }  
    ...  
}
```

The type parameter section (<>), follows the class name.

The type parameters (**T1**, **T2**, etc.) can be used anywhere inside the generic class.

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)

# JAVA GENERICS - GENERIC TYPES 3

**Type Variables:** A type variable can be any non-primitive type:

- any **class** type (example: **GenericPair<String, Integer>**),
- any **interface** type (example: **GenericPair<Comparable<String>, Integer>**),
- any **array** type (example: **GenericPair<int[], Integer>**), or
- any **type variable** (example: **GenericPair<T1, T2>**).

**Note:** The same technique can be used to design **generic interfaces**.

**The Diamond <>:** In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context.

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)

# JAVA GENERICS - GENERIC TYPES 4

## UNCHECKED ERROR MESSAGES

When mixing legacy code with generic code, you may encounter warning messages.

Note: Recompile with `-Xlint: unchecked` for details.

This happens if the compiler does not have the type info to perform all type checks.

```
GenericPair<String,Integer> p = new GenericPair<String,Integer>("A",1); // No warnings.  
GenericPair<String,Integer> p = new GenericPair("A",1); // Unchecked operation.
```

By The "unchecked" warning is disabled, by default.

To see all "unchecked" warnings, recompile with `-Xlint:unchecked`.

```
jGrasp -> Settings -> Compiler Settings -> Workspace -> ...  
... -> Compiler -> Flags/Args -> Compile = -Xlint:unchecked
```

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)

# JAVA GENERICS - GENERIC METHODS

**Generic Methods:** Methods that introduce their own type parameters. In these cases the scope of the type parameters is limited to the method where it is declared.

The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method return type.

```
public static <T1,T2> boolean compare(GenericPair<T1,T2> p1, GenericPair<T1,T2> p2);
```

The complete syntax for invoking this **compare** method would be:

```
boolean res = <Integer,String>compare(pA,pB);
```

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)

# JAVA GENERICS - BOUNDED TYPE PARAMETERS 1

Bounded type parameters **restrict the types** that can be used as type arguments.

To declare a bounded type parameter, list the type parameter, followed by **extends/implements** keywords and its **upper bound** (**String** in this example).

```
public class Box<T1 extends String> { ... }
```

Bounded type parameters also allow you to **call methods defined in the bounds**.

```
T1 a; // T1 (bounded type parameter) is a derived class of String.  
a.substring(...); // Any method in String can be called on T1 (since it derives String).
```

**Note:** A type parameter can have multiple bounds.

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)

# JAVA GENERICS - BOUNDED TYPE PARAMETERS 2

Bounded type parameters are key to the implementation of **generic algorithms**.

**Example:** A method **countGreaterThan** that counts the array items **> e**.

```
public static <T> int countGreaterThan( T[] a, T e ) { ... }
```

This method requires the **> operator** that “applies only” to primitive types.

To fix the issue, use a type parameter bounded by the **Comparable<T>** interface:

```
public static <T extends Comparable<T>> int countGreaterThan( T[] a, T e ) {  
    int count = 0;  
    for( T curr : a ) {  
        if( curr.compareTo(e) > 0 ) { count++; }  
    }  
    return count;  
}
```

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)

# JAVA GENERICS - TYPE INFERENCE

**Type Inference:** The ability of the Java compiler to check each method call and corresponding declaration and decide the type arguments (applicable). The inference algorithm determines the type arguments and (if available) the return type. Then, the inference algorithm finds the most specific types that work with all the arguments.

**Type Witnesses:** Generally, a Java compiler can infer the type parameters of a generic method call. Consequently, in most cases, you do not have to specify them.

**For example:** To call a generic method, you can specify the type parameter:

```
obj.<String>add( s1, s2 ); // Type witness: <String>.
```

If you omit the type witness, the Java compiler infers the type arguments:

```
obj.add( s1, s2 ); // Java compiler infers type arguments from s1 and s2.
```

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)

# JAVA GENERICS - WILDCARDS 1

In generics, the **wildcard** symbol (**?**), represents an **unknown type**.

The wildcard can be used in a variety of situations:

- as the type of a parameter, field, or local variable;
- sometimes as a return type (usually it is better practice to be more specific);
- but, wildcard is **never** used as type argument for method calls, or supertypes.

**Upper Bounded Wildcards:** You can use upper bounded wildcards to relax restrictions on type arguments. To declare an upper bounded wildcard, use: wildcard symbol (**?**), then **extends/implements** keyword, and finally its upper bound class.

```
// This method works on any subclass of Number (using an upper bounded wildcard).  
public static <? extends Number> T count( T[] a, T e ) { ... }
```

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)



# JAVA GENERICS - WILDCARDS 2

**Unbounded Wildcards:** The unbounded wildcard type is specified using the wildcard symbol only (`<?>`), and represent an **unrestricted unknown type**.

There are two scenarios where an unbounded wildcard is a useful approach:

- a method can be coded using just functionality provided by the **Object** class;
- the code using generic methods/types does not depend on the type parameter.

**Lower Bounded Wildcards:** A lower bounded wildcard restricts the type argument to be a specific class or a superclass of that class. A lower bounded wildcard is defined using: wildcard symbol (`?`), then **super** keyword, then its lower bound class.

**Note:** You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)

# JAVA GENERICS - WILDCARDS 3

## Guidelines for Wildcard Use

For this discussion, it is helpful to think of variables as classified as the following:

- **"in" variables:** serve up data to the code (e.g. **src** in method **copy(src, dst)**).
- **"out" variables:** hold data for use elsewhere (e.g. **dst** in method **copy(src, dst)**).

You can use the "in" and "out" principle to decide if and what wildcard is appropriate:

- An "in" variable should be defined with an **upper bounded wildcard**.
- An "out" variable should be defined with a **lower bounded wildcard**.
- If "in" variable can be used with methods in **Object**, use **unbounded wildcard**.
- If a variable is both an "in" and an "out" variable, **do not use a wildcard**.

**Note:** These rules do not apply to return types. Using wildcards for return types should be avoided because it forces coders using the code to deal with wildcards.

**See:** [docs.oracle.com/javase/tutorial/java/generics](https://docs.oracle.com/javase/tutorial/java/generics)

# JAVA EXTRAS - BOXING 1

## BOXING A

Boxing (aka **wrapping**) is the process of placing a **primitive type** (i.e. a **value type**) within an object so that the primitive can be used as a **reference object**.

**Example:** In Java, we cannot create a **LinkedList** of **int**, since this class only lists **references** to objects. To circumvent this, an **int** can be boxed into an **Integer** object, and then added to a **LinkedList** of **Integer** (i.e. a **LinkedList<Integer>**).

**See:** [en.wikipedia.org/wiki/object\\_type\\_\(object-oriented\\_programming\)](https://en.wikipedia.org/wiki/object_type_(object-oriented_programming))

**See:** [docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes)

**See:** [docs.oracle.com/javase/tutorial/reflect/class](https://docs.oracle.com/javase/tutorial/reflect/class)

**See:** [docs.oracle.com/javase/8/docs/api/java/util/LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList)

**See:** [docs.oracle.com/javase/8/docs/api/java/lang/integer](https://docs.oracle.com/javase/8/docs/api/java/lang/integer)

# JAVA EXTRAS - BOXING 2

## BOXING B

**Note:** The boxed object (usually **immutable**) is always a copy of the value object. Unboxing the object also returns a copy of the stored value.

**Note:** Repeated boxing/unboxing decreases performance: it dynamically allocates new objects (boxing) and then marks them for garbage collection (unboxing).

**There is a direct equivalence between an unboxed primitive type and a reference to an immutable boxed object type.**

**See:** [en.wikipedia.org/wiki/object\\_type\\_\(object-oriented\\_programming\)](https://en.wikipedia.org/wiki/object_type_(object-oriented_programming))

**See:** [docs.oracle.com/javase/tutorial/java/data/autoboxing](https://docs.oracle.com/javase/tutorial/java/data/autoboxing)

**See:** [docs.oracle.com/javase/tutorial/essential/concurrency/immutable](https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable)

# JAVA EXTRAS - BOXING 3

## BOXING C

**Example:** We can substitute all primitives in a program with boxed objects. Whereas:

- assignment from one primitive to another will copy its value,
- assignment from one reference to a boxed object to another will copy the reference value to refer to the same object as the first reference.

This is not an issue, because boxed objects are immutable, so there is no semantic difference between 2 references to the same object or to different objects. For all operations other than assignment: unbox the boxed type, perform the operation, and re-box the result as needed. Thus, **it is possible to not store primitive types at all.**

**See:** [en.wikipedia.org/wiki/object\\_type\\_\(object-oriented\\_programming\)](https://en.wikipedia.org/wiki/object_type_(object-oriented_programming))

# JAVA EXTRAS - BOXING 4

## AUTOBOXING

Autoboxing is the term for **getting a reference type out of a value type via type conversion** (implicit or explicit). The Java compiler automatically supplies the extra source code which creates the object.

```
Integer i = new Integer(9); // Allocation of a new Integer object storing the value 9.
Integer i = 9; // Autoboxing: implicit type conversion of value type int (9) into
               //               an Integer object, and assignment of its reference to i.

List<Integer> li = new ArrayList<>();
for( int i = 1; i < 50; i += 2 ) {
    li.add(i); } // At runtime, compiler converts this into: li.add( Integer.valueOf(i) );
```

**See:** [en.wikipedia.org/wiki/object\\_type\\_\(object-oriented\\_programming\)](https://en.wikipedia.org/wiki/object_type_(object-oriented_programming))

**See:** [docs.oracle.com/javase/tutorial/java/data/autoboxing](https://docs.oracle.com/javase/tutorial/java/data/autoboxing)

# JAVA EXTRAS - BOXING 5

## UNBOXING

Unboxing refers to **getting the value which is associated to a given object via type conversion** (implicit or explicit). Java compiler automatically supplies extra code to retrieve the value out of that object (e.g. by invoking some method on that object).

```
Integer a = new Integer(1); Integer b = new Integer(2);  
Integer c = a + b; // Unboxing: a and b are unboxed, their int values summed up,  
                  // and the result is autoboxed into a new Integer object  
                  // (autoboxing via implicit type conversion), and finally  
                  // a reference to this new Integer object is stored in c.
```

**Note:** Operator `==` cannot be used in this way, since it is already defined for reference types (i.e. equality of the references). To test for equality of values of boxed types: unbox boxed objects and compare their unboxed values, or use **equals** method.

# JAVA EXTRAS - BOXING 6

## EXAMPLE AUTOBOXING A

```
// The 8 primitive data types in Java are:  
//     byte: 8-bit signed integer in [ -128, 127 ].  
//     short: 16-bit signed integer in [ -32768, 32767 ].  
//     int: 32-bit signed integer in [ -2^31, 2^31-1 ].  
//         In Java SE 8 and later, can be unsigned 32-bit integer in [ 0, 2^32-1 ].  
//     long: 64-bit integer. The signed long is in [ -2^63, 2^63-1 ].  
//         In Java SE 8 and later, can represent unsigned 64-bit long in [ 0, 2^64-1 ].  
//     float: single-precision 32-bit IEEE 754 floating point.  
//     double: double-precision 64-bit IEEE 754 floating point.  
//     boolean: true or false, and its "size" is not precisely defined.  
//     char: single 16-bit Unicode character in [ '\u0000' or 0, '\uffff' or 65535 ].
```

```
public class ExampleAutoboxing {
```



# JAVA EXTRAS - BOXING 7

## EXAMPLE AUTOBOXING B

```
static public void f( Object o ) {  
    if( o instanceof Byte ) { System.out.println( "Autoboxing in Byte!" ); }  
    else if( o instanceof Integer ) { System.out.println( "Autoboxing in Integer!" ); }  
    else if( o instanceof Long ) { System.out.println( "Autoboxing in Long!" ); }  
    else if( o instanceof Float ) { System.out.println( "Autoboxing in Float!" ); }  
    else if( o instanceof Double ) { System.out.println( "Autoboxing in Double!" ); }  
    else if( o instanceof Boolean ) { System.out.println( "Autoboxing in Boolean!" ); }  
    else if( o instanceof Character ) {  
        System.out.println( "Autoboxing in Character!" ); }  
    else { System.out.println( "Autoboxing in?" ); } }
```

# JAVA EXTRAS - BOXING 8

## EXAMPLE AUTOBOXING C

```
static public void main( String[] args ) {  
    f(-1); // Autoboxing in Integer.  
    f(123); // Autoboxing in Integer.  
    f(5000000000L); // Autoboxing in Long. Call f(5000000000) is autoboxed in Integer.  
    f(123.456); // Autoboxing in Double.  
    f(1.4f); // Autoboxing in Float. A call to f(1.4) mean autoboxing in Double...  
    f(true); // Autoboxing in Boolean.  
    f('a'); // Autoboxing in Character.  
    byte b = -1;  
    f(b); } // Autoboxing in Byte.  
  
    int i = 42;  
    Object o = i; // Autoboxing: value type i autoboxed into reference type Integer.  
    int j = o; // Unboxing error! Incompatible types: Object can't be converted to int.  
}
```

# JAVA EXTRAS - JAVA GC SCHEDULING 1

## The Java Memory Management

Java Memory Management has a built-in Garbage Collector (GC) that allows developers to create new objects without worrying about memory allocation-deallocation, since the GC automatically reclaims unused memory for reuse.

Thanks to the GC, most memory-management issues are solved, but often at the cost of creating serious performance problems. Making garbage collection adaptable to all kinds of situations has led to a complex and hard-to-optimize system.

In order to wrap your head around garbage collection, you need first to understand how memory management works in a Java Virtual Machine (JVM).

**See:** [www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works](http://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works)

# JAVA EXTRAS - JAVA GC SCHEDULING 2

## HOW THE GARBAGE COLLECTION (IN JAVA) WORKS

**You may think the GC collects and discards unused (aka dead) objects.**

**In reality, the GC does the opposite!**

**Used (aka live) objects are tracked and everything else is marked as garbage.**

**The Heap:** is the area of memory used for dynamic allocation. In most configurations the Operating System (OS) allocates the heap in advance to be managed by the Java Virtual Machine (JVM) while the program is running.

**See:** [www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works](http://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works)

# JAVA EXTRAS - JAVA GC SCHEDULING 3

## Using the Heap

Using the heap has a couple of important ramifications:

- Object creation is faster because global synchronization with the OS is not needed for every single object. **An allocation simply claims some portion of a memory array and moves the offset pointer forward.** The next allocation starts at this offset and claims the next portion of the array.
- When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means **there is no explicit deletion and no memory is given back to the OS.**

**See:** [www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works](http://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works)

# JAVA EXTRAS - JAVA GC SCHEDULING 4

All objects are allocated on the heap area managed by the JVM.

Every item that the developer uses is treated this way, including: class objects, static variables, and even the code itself.

As long as an object is being referenced, the JVM considers it alive.

Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory.

**See:** [www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works](http://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works)

